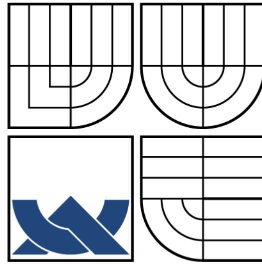


Brno University of Technology
Università degli Studi dell'Aquila

Mathematical Engineering

Laurea Specialistica in Ingegneria Matematica

Heuristic Algorithms in Optimization

Author:	Čeněk Šandera
Supervisor:	Jan Roupec
Project Supervisor:	Bruno Rubino

2007/2008

Acknowledgement

First and foremost, I would like to thank my whole family because this thesis would have never been complete without their unlimited support and never-ending belief. My great thank also goes to my supervisor Jan Roupec who has opened me the door to the astonishing world of heuristic methods and to my consultant Pavel Popela for his enthusiasm during our discussions about stochastic programming.

Statement

I declare that this is an original thesis and is entirely my own work and wherever I used the ideas of other writers, I acknowledge the source in every instance.

Čeněk Šandera

Abstract

This thesis deals with a stochastic programming and determining probability distributions which cause extreme optimal values (maximal or minimal) of an objective function. The probability distribution is determined by heuristic method, especially by genetic algorithm, where whole population approximates the desired distribution. The first parts of the thesis describe mathematical and stochastic programming in general and also there are described various heuristic methods with emphasis on genetic algorithms. The goal of the diploma thesis is to create a program which tests the algorithm on linear and quadratic stochastic models.

Contents

Introduction vi

1 Stochastic programming 1
1.1 Optimization 1
1.1.1 General mathematical formulation 1
1.1.2 Linear programming 2
1.1.3 Quadratic programming 3
1.2 Stochastic programming 4
1.2.1 General mathematical formulation 4
1.2.2 Linear two-stage stochastic programming 6
1.2.3 Quadratic two-stage stochastic programming 7
1.2.4 Solving the stochastic programs 8
1.2.5 Monte Carlo approximation of stochastic programs 8
1.3 An analysis of extreme cases 8
1.3.1 Determination of a confidence interval 9
1.3.2 Minimax approach 12

2 Heuristic methods 14
2.1 Hill Climbing Algorithm 15
2.2 Tabu search 15
2.3 Simulated annealing 16
2.4 Ant colony optimization 16
2.5 Bees optimization 17
2.6 Evolutionary Algorithms 17
2.6.1 Genetic algorithms 19
2.7 Various implementations of genetic algorithms 22
2.7.1 Representation of chromosomes 22
2.7.2 Size of populations 23
2.7.3 Initial population 24
2.7.4 Fitness function 24
2.7.5 Parent selection 24
2.7.6 Crossover 25
2.7.7 Mutation 26
2.7.8 Integration offsprings into a population 26
2.7.9 Termination criteria 27

3 Algorithms for determining boundaries of stochastic programming models 28
3.1 Classical genetic algorithm 28
3.2 An approach based on modified genetic algorithm 29

4 Testing the algorithm 31
4.1 Models 31
4.1.1 Linear stochastic two-stage model 31
4.1.2 Quadratic stochastic two-stage model 33
4.2 Monte Carlo investigating of confidence interval 35
4.3 Distribution of the fitness functions in the whole populations 37
4.4 Evaluating of populations 40
4.5 Computer implementation 42
4.6 Algorithm for searching minimal distributions 43
4.7 Algorithm for searching the maximal distributions 46

5 Conclusion 49

A Distributions of chromosome fitnesses 50

B Data for evaluating populations 51
B.1 Model I 51
B.2 Model IV 52

C Searching for a minimum 54
C.1 Evolution of the objective function 54
C.2 Evolution of distributions 55
C.3 Final distributions with minimal value 56

D Searching for the maximum 57
D.1 Evolution of the searching for maximal value 57
D.2 Evolution of maximal distributions 58
D.3 Final distributions with maximal value 59

Index 60

Introduction

The main topic of this diploma thesis are heuristic algorithms. Heuristic algorithms are amazing artificial instruments which allow us to deal with the most complicated and most challenging problems of the present days. A special subclass of these algorithms inspirited by Darwin's Theory of Evolution is called genetic algorithms and it has many interesting application. Engineers, scientists and designers, all over the world, are engaged every day in problems of determining the best possible configuration for their machines, theories or inventions. Genetic algorithms can help with a very large class of problems which are solved by classical analytical methods very hardly. The main difference between classical and heuristic method is in expected solution. Every man who solve arbitrary problem wants to obtain the best possible solution but in the real life there are usually sufficient to know a solution which is just very close to the optimal one and this is exactly the place where heuristic methods rule. Nobody can assure an obtaining even very good solution by this way but there is empirically proved that there exists some special heuristic method for every kind of optimization problem.

In this thesis we apply the genetic algorithms to problems of stochastic programming. The stochastic programming an is interesting way how to define and solve optimization problems with random coefficients. Every process in the world is influenced by great amount of various random events and if we want to model whichever real system we have to somehow deal with these randomnesses. Many times the random influences are omitted or considered to be neglected but this way is not the right one in general. Stochastic programming is designed for problems where the results, obtained by classical mathematical programming, are incorrect and can let us to make completely wrong decisions. It is also very useful to know an impact of the randomness in the investigated model and to know which series of random events is the most positive and which one is the worst for us. So the thesis deals with determining these boundaries and investigating how the optimal solution of stochastic programs is changed under different probability distributions. There will be implemented a special genetic algorithm which can give us the answers for all these questions.

The outcome of the thesis is to prepare an implementation of the genetic algorithm and apply it to several different mathematical models. This algorithm is created in programming language Python and it was chosen because of its property to easily extend and change the program's structure. Genetic algorithms have many different forms and settings and the thesis' outcome is also to find the most appropriate implementation for given problems.

1 Stochastic programming

1.1 Optimization

Optimization or *mathematical programming* is part of mathematics which deals with finding the best possible solution of prescribed function under the given conditions. This concept has many applications, for instance, in distribution of goods and resources, engineering systems design, financial planning, manpower and resource allocation, manufacturing of goods, or in sequencing and scheduling of tasks. Many optimization problems are very large and the time needed for a testing all possible solutions, even on contemporary supercomputers, can take years of computing power. This is the reason why many mathematicians focus on a development of algorithms which can dramatically reduce computing time.

1.1.1 General mathematical formulation

After studying many optimization problems we can say that most of the optimization problem have similar structure. So we can generalize them and write in proper mathematical form. The main part of this form is so-called *objective function*. This function describes how quality the solution is. The goal of a optimization procedure is to find a solution which gives extreme value of this objective function. The extremes can be either maximal or minimal objective value and each problem of finding a maximum can be simply transformed to a problem of finding a minimum and vice versa, therefore, for simplification, in the further text we consider just problem to determine minimum value of an objective function. A domain of an objective function is called *search space* and is restricted by number of *constraints*. There are either equality or inequality constraints and each solution has to satisfy all of them. The set of solutions which satisfy all constraints are called *feasible set* and its members are so called *candidate solutions*. The general mathematical formulation is

$$\left.\begin{array}{r} \min\limits_{x \in X}\{f(\boldsymbol{x})\} \\ \text{subject to:} \\ g_i(\boldsymbol{x}) \leq 0 \end{array}\right\} \tag{1.1}$$

Where $f : \mathbb{R}^n \rightarrow \mathbb{R}$ is an objective function, $\boldsymbol{x}$ is n-dimensional variable vector, $g_i : \mathbb{R}^n \rightarrow \mathbb{R}$, for $i = 1, \ldots, m$, are constraints which define feasible space X.

Alternatively instead of $\min_{x \in X}\{f(\boldsymbol{x})\}$ we can use $\boldsymbol{x}_{min} \in \text{argmin}_{x \in C}\{f(\boldsymbol{x})\}$[1] to emphasise that we are not looking just for an extremal value of objective function, but we mainly need to know values of variables which this extreme caused.

[1] $\text{argmin}_{x \in X}\{f(x)\} \in \{x \in X | f(x) < f(y), \forall y \in X\}$

If all variables are required to be integers then the problem is called *integer programming*. A special class of integer programming is *binary integer programming* where the unknown variables has to be numbers 0 or 1.

1.1.2 Linear programming

A special case of mathematical programming is *linear programming*. All equations and inequalities in mathematical model of the problem have to be linear with respect to the variables representing a solution. Each linear program can be expressed in this canonical form

$$\left.\begin{array}{l} \min\limits_{(x_1,x_2,\ldots,x_n)\in X} \{c_1x_1 + c_2x_2 + \ldots + c_nx_n\} \\ \text{subject to:} \\ \quad \begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \cdots & + & a_{1n}x_n & \leq & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \cdots & + & a_{2n}x_n & \leq & b_2 \\ & & \vdots & & & & & & \vdots \\ a_{m1}x_1 & + & a_{m2}x_2 & + & \cdots & + & a_{mn}x_n & \leq & b_m \end{array} \end{array}\right\} \tag{1.2}$$

this notation can be simplified by using matrix and vectors

$$\left.\begin{array}{l} \min\limits_{x\in X}\{\boldsymbol{c}^{\mathrm{T}}\boldsymbol{x}\} \\ \text{subject to:} \\ \quad A\boldsymbol{x} \leq \boldsymbol{b} \\ \quad \boldsymbol{x} \in X \subset \mathbb{R}^n \end{array}\right\} \tag{1.3}$$

Where $\boldsymbol{c}^{\mathrm{T}} \in \mathbb{R}^n$ is a transpose constant n-dimensional vector, $\boldsymbol{x}$ is vector of n variables, A is a constant matrix $[a_{ij}]$ for $i = 1,\ldots,m$ and $j = 1,\ldots,n$ and $\mathbf{b}$ is m-dimensional constant vector represent model's constraints.

Linear models have many useful properties which make their solution easier. It is one of the most studied topic in the mathematical programming, many real-life problems can be transformed into an appropriate linear form and solved by some of known efficient algorithms. The most known numerical algorithms is *Simplex algorithm* cite[17634] developed by George Dantzig[2] in 1947. This algorithm is based on very important property called convexity. Each feasible space of a linear programming problem can be geometrically represented as *convex polyhedron* and the extreme values (maximum or minimum) of the objective function can arise only in one of its vertices. The simplex algorithm therefore just numerically looks for the vertices and checks if there is an extreme of the function. Linear models assure that if an algorithm finds any point of local extreme than it is the point of global extreme also. Because there is just one point of local maximum or local minimum in linear programming and this point clearly has to be the global extreme also. This properties dramatically reduce a number of tested points and allow us to solve models with thousands of variables and constraints.

[2] American mathematician, (* November 8th, 1914 - †May 13th, 2005)

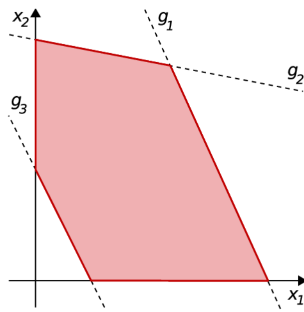


Figure 1.1 Example of a feasible set in a linear programming

The Figure 1.1 shows an example of graphical expression of two-dimensional linear programming. The red area is convex polyhedron which represent a set of candidate solutions and it is restricted by three linear conditions g_1, g_2, g_3 and by requirements $x_1 \geq 0$ and $x_2 \geq 0$. The area is a feasible set and has to contain the desired solution of the problem.

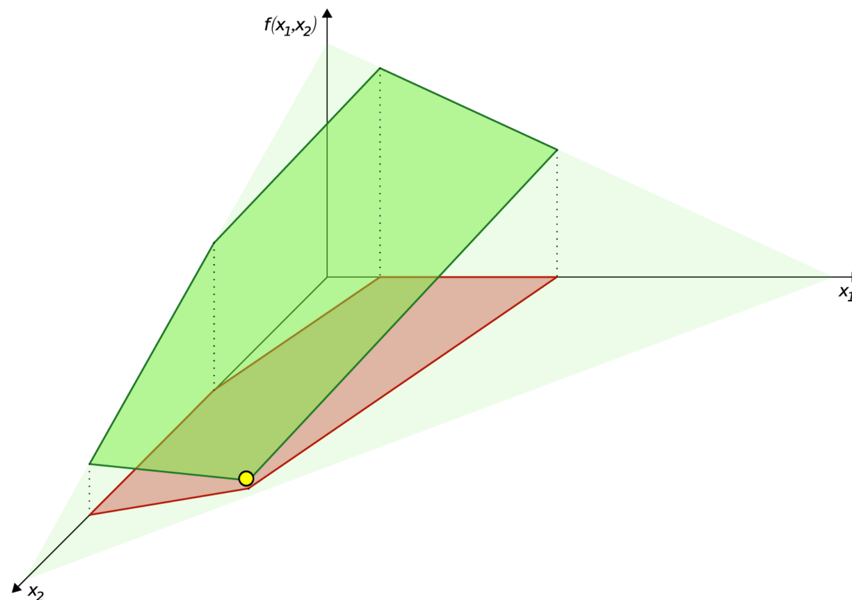


Figure 1.2 Example of an objective function in linear programming

The Figure 1.2 shows the feasible set from the figure 1.1 with an example of a linear objective function $f(x_1, x_2)$. It is easy to see that because of the linearity the extreme value (the yellow point) has to occur exactly in one of the corners of the feasible set.

1.1.3 Quadratic programming

If the objective function is polynomial of second degree and constrains are linear than the optimization problem is called *quadratic programming*. This model has many application, for instance,

widely used portfolio investment strategy by Harry Markowitz[3] [22] and because of its importance there exist several efficient algorithms for exact solving. The general mathematical formulation for quadratic programming is

$$\left.\begin{array}{l} \min\limits_{x \in X} \left\{ \dfrac{1}{2} \boldsymbol{x}^{\mathrm{T}} H \boldsymbol{x} + \boldsymbol{c}^{\mathrm{T}} \boldsymbol{x} \right\} \\ \text{subject to:} \\ \quad A\boldsymbol{x} \leq \boldsymbol{b} \\ \quad \boldsymbol{x} \in X \subset \mathbb{R}^n \end{array}\right\} \tag{1.4}$$

where $\boldsymbol{x}$ is a variable vector, $\boldsymbol{c}$ is a constant cost of the linear part and H is a constant symmetric matrix which determines a cost of the quadratic part of the model.

A behaviour of the model highly depends on character of matrix H. The easiest case of quadratic programming arises if the matrix H is positive definite (or negative definite for maximization problem). In this case the objective function is convex and the optimal solution is unique. The Markowitz portfolio optimization is usually of this type. If H is semi-definite than the objective function is still convex, but solution cannot be unique. In case with H indefinite the objective function has a saddle and therefore is non-convex. This type of quadratic programming is the most difficult one and its solution takes a lot of time by specialized quadratic solvers.

1.2 Stochastic programming

Usually mathematical models are built as representation of some real life problem and an objective function or constraints are determined by some observations or experiments. Therefore coefficients used in mathematical model like this are set to some typical values and the model is called *deterministic model*. Often it is sufficient and a model built by this way describes system behaviour very precisely. But there is a large class of problems which cannot be modelled just by using static coefficients and we have to consider influence of uncertainty. This concept is called *stochastic programming* and typical application can be found in financial planning, where these models can describe trends for stocks and they help to determine how much should be invested. Another applications are, for instance, in biology for modelling animal behaviour or in engineering for optimizing dimensions of designed products. Stochastic programs are much more complicated for solving than corresponding deterministic problems, so there should be some good reason to choose the stochastic form, but many times we don't know if deterministic models are sufficient to describe a problem properly , and therefore the only way how to model it is to use the stochastic programming.

1.2.1 General mathematical formulation

The uncertainty in mathematical programming is modelled by theory of random variables. Each variable can has arbitrary distribution of probability and can occur wherever in the model. Very detailed description can be found in [5]. So generally we have

[3] American economist, Nobel prize 1990, (* August 24th, 1927)

$$\left.\begin{array}{ll}\min\limits_{x \in X(\tilde{\xi})} \{f(\boldsymbol{x}, \tilde{\xi})\} & \\ \text{subject to:} & \\ \quad g_i(\boldsymbol{x}, \tilde{\xi}) \leq 0, & \text{for } i = 1, \ldots, m \\ \quad \boldsymbol{x} \in X(\tilde{\xi}) \subset \mathbb{R}^n & \text{almost surely}\end{array}\right\} \tag{1.5}$$

Where $\tilde{\xi}$ is random vector defined on the probability space $(\Xi, \Sigma, \mathrm{P})$, objective function $f : \Xi \times \mathbb{R}^n \rightarrow \mathbb{R}$ is measurable function for each decision $\boldsymbol{x} \in \mathbb{R}^n$ which has to belong to feasible set X.

There are several ways how to understand the goal of optimization. Clearly, the best value of an objective function is reached if we wait for values of random variables and optimize model according to them. This kind of optimizing is called *wait-and-see*, but unfortunately, we have to usually optimize model before realizations of random events, so what we actually want is minimize objective functions for each possible realization. In the other words, model is changed to finding minimal expectation value of an objective function under the given random variable. This solution is called *here-and-now*.

$$\left.\begin{array}{ll}\min\limits_{x \in X(\tilde{\xi})} \{E_{\xi} f(\boldsymbol{x}, \tilde{\xi})\} & \\ \text{subject to:} & \\ \quad g_i(\boldsymbol{x}, \tilde{\xi}) \leq 0, & \text{for } i = 1, \ldots, m \\ \quad \boldsymbol{x} \in X(\tilde{\xi}) \subset \mathbb{R}^n & \text{almost surely}\end{array}\right\} \tag{1.6}$$

The fundamental idea behind the stochastic programming is a *recourse problem*. We can divide the model into two parts. The first part is before taking a place of random event and the second one is after that. In the first part (so-called *first stage*) we don't know how the model is going to behave and we have to take a decision under uncertainty. In the second part (so-called *second stage*) taking a decision is much easier because we have already observed random events and we know exact values of them. A decision taken after observing random events is called *recourse* and its purpose is to correct mistakes and inaccuracies caused by decision taken in the first stage. In fact, the second stage is just a deterministic model without any uncertainties. The goal of optimizing models like this is to find values for the first stage which minimize objective function for each recourse. In mathematical way, the second stage is replaced by expectation value of recourse function under the given probability distribution. The described model is called *two-stage model with recourse* and can be generalized into *multi-stage model*. In multi-stage model random events don't take place in one time and before each set of events some decision has to be taken. So the general formula of multistage stochastic program with recourse is

$$\min_{x_0 \in X(\tilde{\xi})} \{q_0(\boldsymbol{x}_0) + \sum_{\tau=1}^{K} E_{\tilde{\xi}_1, \ldots, \tilde{\xi}_\tau} Q_\tau(\boldsymbol{x}_0, \hat{\boldsymbol{x}}_1, \ldots, \hat{\boldsymbol{x}}_{\tau-1}, \tilde{\xi}_1, \tilde{\xi}_2, \ldots, \tilde{\xi}_\tau)\}, \tag{1.7}$$

where Q_τ is a recourse function at stage $\tau \geq 1$ given by

$$Q_\tau(\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{\tau-1}, \xi_1, \ldots, \xi_\tau) = \min_{\boldsymbol{x}_\tau} \{q_\tau(\boldsymbol{x}_\tau) | g_\tau(\boldsymbol{x}_0, \ldots, \boldsymbol{x}_\tau, \xi_1, \ldots, \xi_\tau) \leq 0\}$$

indicating that the optimal recourse action $\hat{\boldsymbol{x}}_\tau$ at time τ depends on the previous decisions and realizations ξ_i observed before the stage τ, i.e.

$$\hat{\boldsymbol{x}}_\tau = \hat{\boldsymbol{x}}_\tau(\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{\tau-1}, \xi_1, \ldots, \xi_\tau),\ \tau \geq 1$$

1.2.2 Linear two-stage stochastic programming

An important special case of a stochastic programming is *linear two-stage stochastic programming model with recourse*. This model is combination of deterministic linear model and stochastic two-stage model. Mathematical formulation of *stochastic linear program with fixed recourse* is

$$\left.\begin{array}{l} \min\limits_{x \in X} \{\boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} + E_{\tilde{\xi}}(Q(\boldsymbol{x}, \tilde{\xi}))\} \\ \text{subject to:} \\ \quad A\boldsymbol{x} \leq \boldsymbol{b} \\ \quad \boldsymbol{x} \geq \boldsymbol{0} \\ \text{where} \\ \quad Q(\boldsymbol{x}, \xi) = \min\limits_{y}\{\boldsymbol{q}^T\boldsymbol{y} \mid W\boldsymbol{y} \geq h(\xi) - T(\xi)\boldsymbol{x},\ \boldsymbol{y} \geq 0\} \end{array}\right\} \tag{1.8}$$

where $\tilde{\xi}$ is a random variable and ξ is its one concrete observation. The function $Q(\boldsymbol{x}, \tilde{\xi})$ represents the second-stage decision.

In particular, if the second stage program is always feasible, than we speak about *complete fixed recourse* given by *fixed recourse matrix W*.

Linear stochastic models have some useful properties, like deterministic linear models, and therefore it is very often studied and applied area of stochastic programming models.

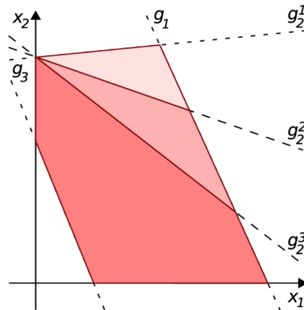


Figure 1.3 An example of a feasible set with stochastic linear constraints

The Figure 1.3 shows an example of graphical expression of two-dimensional stochastic programming. The feasible set (the red area) is given by three linear constraints g_1, g_2^i, g_3 and requirements $x_1 \geq 0$ and $x_2 \geq 0$. The model contains one random variable which can have three different realizations with the same probabilities. Each random value determines different constrain g_2^i and therefore there are three possible feasible sets which have to be taken into account.

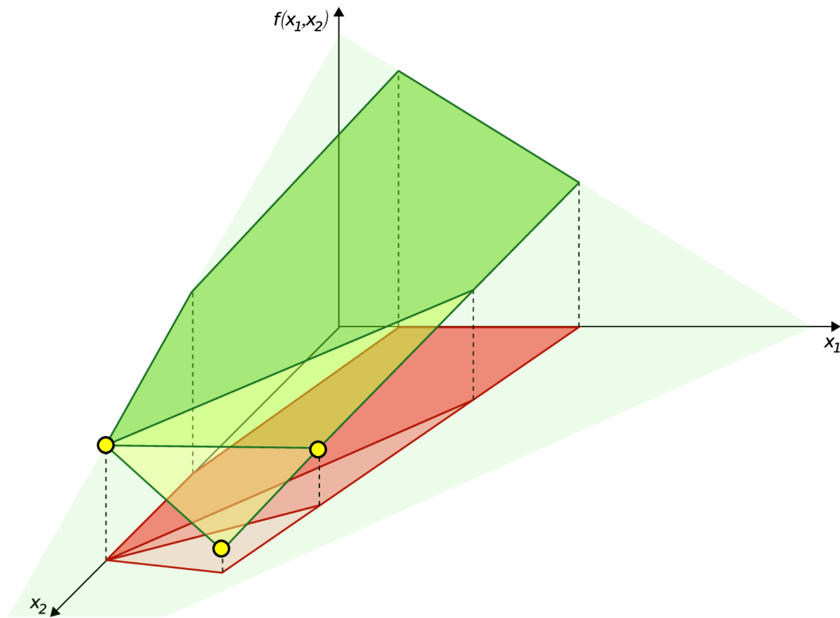


Figure 1.4 An example of an objective function in stochastic linear programming

The Figure 1.4 shows the feasible sets from the Figure 1.3 with an example of a linear objective function $f(x_1, x_2)$. For each feasible set with the same objective function there exists different minimal value highlighted by the yellow points. This example shows complicatednesses of solving stochastic programs in general, because just one random parameter can change solutions and optimal values very dramatically.

1.2.3 Quadratic two-stage stochastic programming

Another important stochastic model is derived from quadratic programming. Objective functions in the both stages are quadratic and its constrains are linear.

$$\left.\begin{array}{l} \min\limits_{\boldsymbol{x}\in X} \left\{\frac{1}{2}\boldsymbol{x}^{\mathrm{T}}H\boldsymbol{x} + \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} + E_{\tilde{\xi}}(Q(\boldsymbol{x}, \tilde{\xi}))\right\} \\ \text{subject to:} \\ \quad A\boldsymbol{x} \leq \boldsymbol{b} \\ \quad \quad \boldsymbol{x} \geq \boldsymbol{0} \\ \text{where} \\ \quad Q(\boldsymbol{x}, \xi) = \min\limits_{\boldsymbol{y}} \left\{\frac{1}{2}\boldsymbol{y}^{\mathrm{T}}G\boldsymbol{y} + \boldsymbol{q}^{T}\boldsymbol{y} | \, W\boldsymbol{y} \geq h(\xi) - T(\xi)\boldsymbol{x}, \, \boldsymbol{y} \geq 0\right\} \end{array}\right\} \tag{1.9}$$

The H and G are symmetric matrices which determine coefficients for quadratic members in the objective function.

1.2.4 Solving the stochastic programs

A solution of a stochastic program is highly dependent on the probability distribution of random events. If we consider discrete distribution of two-stage programs with just a few possible realizations we can transform the problem to exact *deterministic equivalent form* and compute the solution by using classical algorithms of mathematical programming. For instance, let's suppose the random variable $\tilde{\xi}$ can assume just three different values (ξ_1, ξ_2, ξ_3) with given probabilities (p_1, p_2, p_3).

Hence the stochastic programming problem

$$\min_{x \in X} \{E_{\tilde{\xi}}(f(\boldsymbol{x}, \tilde{\xi}))\} \tag{1.10}$$

can be transformed into deterministic equivalent form

$$\min_{x \in X} \{p_1 f(\boldsymbol{x}, \xi_1) + p_2 f(\boldsymbol{x}, \xi_2) + p_3 f(\boldsymbol{x}, \xi_3)\} \tag{1.11}$$

Evident drawback of this approach is a growing size of equivalent problem for random values with many different realizations. Even for a few tens of realizations the equivalent deterministic model can be unsolvable in acceptable time. Moreover random variables with continuous distribution function cannot be expressed by this way and involve computing of n-dimensional integral for the exact solution of expectation value. These drawbacks lead us to another method to determine the optimal solution.

1.2.5 Monte Carlo approximation of stochastic programs

Because the exact solution of stochastic models involves a lot of computing resources we should focus on approximation methods. Well known approximation is *Monte Carlo method*. It is based on a simple idea to approximate continuous distribution by a few randomly generated scenarios. By one *scenario* is meant one set of realizations of the random variables occur in the model. One scenario completely determine behaviour of the system in one specific combination of random events. Each scenario has its own probability and if the random events are independents then the probability is given as product of all member's probabilities contained in the set. So in the Monte Carlo method we just need to know how large problems we are able to solve and than by n randomly generated scenarios ξ_i solve the *approximating problem*

$$\min_{x \in X} \{p_1 f(\boldsymbol{x}, \xi_1) + p_2 f(\boldsymbol{x}, \xi_2) + \ldots + p_n f(\boldsymbol{x}, \xi_n)\}. \tag{1.12}$$

By repetition of this approach we can eliminate random deviations and more precisely estimate the real expectation value.

1.3 An analysis of extreme cases

If we deal with a problem of stochastic programming and we want to make a decision based on our computation we usually need to know as many information about randomness contained in our

model as possible. Unfortunately there exist many problems with random coefficients where we know almost nothing about their distribution function, variance, expectation value or any other useful statistic characteristic. Many engineers have to design their products with given reliability, power plants need to be prepared for extreme demands and bank managers need to predict the biggest losses on the stock market. Random influences in these problems are hardly described by any of the aforementioned statistic characteristic and the only what we can usually do is to estimate some of them by expert analysis. So if an engineer, a power plant or a bank manager wants to determine what is the worst random influence for his model then he can just to find a set of scenarios with the worst objective function value to approximate the worst probability distribution which can occur. Another approach can be determine confidence interval for objective function values as a lower and upper bound with certain probability.

1.3.1 Determination of a confidence interval

For random variable $\tilde{\xi}$ defined on probability space $(\Xi, \Sigma, \mathrm{P})$, the *confidence interval* is interval $\langle a, b\rangle$ which contains prescribed amount of its realizations. Therefore confidence interval satisfies equality

$$\mathrm{P}(a < \xi < b) = 1 - \alpha,$$

where P is a probability function, ξ is realization of random variable $\tilde{\xi}$ and the expression $1-\alpha$ is called *confidence coefficient*. The number α, so-called *confidence level*, can be also interpreted as probability of being realization ξ out of the interval.

For determining the confidence interval of an objective function in stochastic programs, Mak, Morton and Wood proposed [3] relatively simple computational approach based on the Monte Carlo algorithm.

Let's consider stochastic optimization problem in the form

$$z^* = \min_{x \in X} \left\{ E f(\boldsymbol{x}, \tilde{\xi}) \right\}$$

with

$$x^* \in \operatorname*{argmin}_{x \in X} \left\{ E f(\boldsymbol{x}, \tilde{\xi}) \right\},$$

where f is the objective function, x is vector if decision variables from feasible set X. Also we introduce associated *approximating problem* SP_n

$$z_n^* = \min_{x \in X} \left\{ \frac{1}{n} \sum_{i=1}^{n} f(\boldsymbol{x}, \tilde{\xi}^i) \right\}$$

with

$$x_n^* \in \underset{x \in X}{\operatorname{argmin}} \left\{ \frac{1}{n} \sum_{i=1}^{n} f(\boldsymbol{x}, \tilde{\xi}^i) \right\},$$

where $\tilde{\xi}^i$, $i = 1, \ldots, n$ are independent identically distributed (i.i.d.) random variables from the distribution of $\tilde{\xi}$.

By solving approximating deterministic problem in the form SP_n we get *candidate solution* $\hat{\boldsymbol{x}}$ and we can define *optimality gap* as $Ef(\hat{\boldsymbol{x}}, \tilde{\xi}) - z^*$. In fact, the optimality gap is the difference in objective functions between a candidate solution and the optimal solution. This method requires only mild assumptions: $f(\boldsymbol{x}, \tilde{\xi})$ has finite mean and variance, i.i.d. observations of $\tilde{\xi}$ can be generated, instances of SP_n can be solved for sufficiently large n to yield "good" bounding information, and $f(\boldsymbol{x}, \tilde{\xi})$ can be evaluated exactly for specific values of $\boldsymbol{x}$ and realizations of $\tilde{\xi}$.

Upper bounds

Suppose we can find a good but suboptimal solution $\hat{\boldsymbol{x}} \in X$ for stochastic program SP. Therefore we can estimate $Ef(\hat{\boldsymbol{x}}, \tilde{\xi})$ via the standard sample mean estimator

$$\bar{U}(n) = \frac{1}{n} \sum_{i=1}^{n} f(\hat{\boldsymbol{x}}, \tilde{\xi}^i), \tag{1.13}$$

where $\tilde{\xi}^1, \ldots, \tilde{\xi}^n$ are i.i.d. from the distribution $\tilde{\xi}$. This estimator is unbiased estimator of the true value of a suboptimal solution $\hat{\boldsymbol{x}}$, i.e. the upper bound can be expressed as

$$E\bar{U}(n) = Ef(\hat{\boldsymbol{x}}, \tilde{\xi}) \geq z^*,$$

and another important property of the estimator follows from the Central limit theorem.

Theorem 1 (Central limit theorem)

Let $X_1, X_2, \ldots, X_n$ be a set of n independent and identically distributed random variables having finite values of mean μ and variance $\sigma^2 > 0$. And let the S_n be a sum of n random variables given by $S_n = X_1 + X_2 + \ldots + X_n$, now we can define

$$Z_n = \frac{S_n - n\mu}{\sigma\sqrt{n}}.$$

Than the distribution of Z_n converges in distribution towards the standard normal distribution $\mathcal{N}(0,1)$ as n increase to infinity. This means that for every $z \in \mathbb{R}$

$$\lim_{n\to\infty} \mathrm{P}\left(\frac{\overline{X}_n - \mu}{\sigma/\sqrt{n}} \leq z \right) = \Phi(z),$$

where $\Phi(z)$ is cumulative distribution function of $\mathcal{N}(0,1)$ and $\overline{X}_n = S_n/n = (X_1 + X_2 + \ldots + X_n)/n$ is the sample mean.

Thus by using the Central limit theorem we can deduce

$$\sqrt{n}[\bar{U}(n) - Ef(\hat{\boldsymbol{x}}, \tilde{\xi})] \Rightarrow \mathcal{N}(0, \sigma_u^2) \text{ as } n \to \infty$$

where $\sigma_u^2 = \text{var} f(\hat{\boldsymbol{x}}, \tilde{\xi})$.

Lower bound

For lower bounds of the optimal solution z^* we can use identically independent distributed random variables $\tilde{\xi}^1, \tilde{\xi}^2, \dots, \tilde{\xi}^n$ from the distribution $\tilde{\xi}$ and we can say

$$Ez_n^* = E \min_{x \in X} \left[\frac{1}{n} \sum_{i=1}^{n} f(\boldsymbol{x}, \tilde{\xi}^i) \right] \leq z^*.$$

This inequality comes from

$$z^* = \min_{x \in X} Ef(\boldsymbol{x}, \tilde{\xi}) = \min_{x \in X} E \frac{1}{n} \sum_{i=1}^{n} f(\boldsymbol{x}, \tilde{\xi}^i) \geq E \min_{x \in X} \frac{1}{n} \sum_{i=1}^{n} f(\boldsymbol{x}, \tilde{\xi}^i)$$

and together with aforementioned upper bound and batch processing it can give us a confidence interval.

A confidence interval construction

Let $\tilde{\xi}^{i1}, \dots, \tilde{\xi}^{in}$ be i.i.d. batches of random vectors, for $i = 1, \dots, n_l$. We define an optimal value of approximating problem with ith batch of random variable as

$$z_n^{*i} = \min_{x \in X} \frac{1}{n} \sum_{j=1}^{n} f(\boldsymbol{x}, \tilde{\xi}^{ij}).$$

and an estimate for lower bound based on approximated optimal values

$$\bar{L}(n_l) = \frac{1}{n_l} \sum_{i=1}^{n_l} z_n^{*i}. \tag{1.14}$$

Than according to the Central limit theorem we have

$$\sqrt{n_l}\,[\bar{L}(n_l) - Ez_n^*] \Rightarrow \mathcal{N}(0, \sigma_l^2) \text{ as } n_l \to \infty$$

where $\sigma_l^2 = \text{var} z_n^*$.

Let $t_{n-1,\alpha}$ satisfy $\mathrm{P}(T_n \leq t_{n-1,\alpha}) = 1 - \alpha$, where the random variable T_n has a t-distribution with $n-1$ degrees of freedom. Let $s_l^2(n_l)$ denotes the standard sample variance estimator of σ_l^2, let n_u be a number of observations used to estimate $Ef(\hat{\boldsymbol{x}}, \tilde{\xi})$, and define

$$\tilde{\varepsilon}_l = \frac{t_{n_l-1,\alpha} s_l(n_l)}{\sqrt{n_l}}$$

hence the confidence interval for estimating $E\bar{L}(n_l)$ is $[\bar{L}(n_l) - \tilde{\varepsilon}_l, \bar{L}(n_l) + \tilde{\varepsilon}_l]$ and analogically for estimating $E\bar{U}(n_u)$ we defined

$$\tilde{\varepsilon}_u = \frac{t_{n_u-1,\alpha} s_u(n_u)}{\sqrt{n_u}}$$

and the confidence interval is therefore $[\bar{U}(n_u) - \tilde{\varepsilon}_l, \bar{U}(n_u) + \tilde{\varepsilon}_u]$. The upper bound estimator is computed using stream of observation $\tilde{\xi}^1, \dots, \tilde{\xi}^{n_u}$ and is independent on the stream used for the lower bound estimate. By using all of these facts and Boole's inequality we have

$$\begin{aligned}
&\mathrm{P}\left(\bar{L}(n_l) - \tilde{\varepsilon}_l \le Ez_n^* \le z^* \le Ef(\hat{\boldsymbol{x}}, \tilde{\xi}) \le \bar{U}(n_u) + \tilde{\varepsilon}_u\right) = \mathrm{P}(\{\bar{L}(n_l) - \tilde{\varepsilon}_l \le Ez_n^*\} \\
&\quad \cap \{Ez_n^* \le z^*\} \cap \{z^* \le Ef(\hat{\boldsymbol{x}}, \tilde{\xi})\} \cap \{Ef(\hat{\boldsymbol{x}}, \tilde{\xi}) \le \bar{U}(n_u) + \tilde{\varepsilon}_u\}) \\
&= 1 - \mathrm{P}(\{\bar{L}(n_l) - \tilde{\varepsilon}_l > Ez_n^*\} \cap \{Ez_n^* > z^*\} \cap \{z^* > Ef(\hat{\boldsymbol{x}}, \tilde{\xi})\} \\
&\quad \cap \{Ef(\hat{\boldsymbol{x}}, \tilde{\xi}) > \bar{U}(n_u) + \tilde{\varepsilon}_u\}) \\
&\ge 1 - \mathrm{P}(\bar{L}(n_l) - \tilde{\varepsilon}_l > Ez_n^*) - \mathrm{P}(Ez_n^* > z^*) - \mathrm{P}(z^* > Ef(\hat{\boldsymbol{x}}, \tilde{\xi})) \\
&\quad - \mathrm{P}(Ef(\hat{\boldsymbol{x}}, \tilde{\xi}) > \bar{U}(n_u) + \tilde{\varepsilon}_u) \\
&\approx 1 - \alpha - 0 - 0 - \alpha = 1 - 2\alpha
\end{aligned}$$

Therefore, the formula

$$[\bar{L}(n_l) - \tilde{\varepsilon}_l, \bar{U}(n_u) + \tilde{\varepsilon}_u] \tag{1.15}$$

is $(1 - 2\alpha)$-level confidence interval for the objective function values of the investigated problem.

We can also construct $(1 - 2\alpha)$-level confidence interval for an optimality gap at $\hat{\boldsymbol{x}}$ in the form

$$[0, \bar{U}(n_u) - \bar{L}(n_l) + \tilde{\varepsilon}_l + \tilde{\varepsilon}_u]\,.$$

Due to sampling error we may actually observe $\bar{U}(n_u) < \bar{L}(n_l)$ therefore for more conservative confidence interval is better to use

$$[0, [\bar{U}(n_u) - \bar{L}(n_l)]^+ + \tilde{\varepsilon}_l + \tilde{\varepsilon}_u]\,,$$

where $[y]^+ \equiv \max\{y, 0\}$.

Now we are able to estimate boundaries of an objective function for stochastic programming problem. Because the computation of the boundaries are based on statistical methods the resultant interval is given with certain probability. It means, the boundaries do not cover all values of an objective function, but just certain part given by a chosen confidence level α.

1.3.2 Minimax approach

Another approach how to determine boundaries for stochastic program with unknown probability distribution is so-called *minimax approach*. It is based on a finding probability distribution which realizes an extreme of the following model

$$\min_{x \in X} \max_{P \in \mathcal{P}} f(\boldsymbol{x}, P),$$

where $\mathcal{P}$ is a set of all "appropriate" distributions. Even though, we don't know an exact form of the probability distribution we can sometimes have information about its support, about values of some its moments or about any other its characteristics. Therefore the $\mathcal{P}$ is set of probability distributions which satisfy all of these known properties.

There exist a few results for special stochastic models which simplify or even exactly solve the minimax problem. For instance, according to [4], if we consider a model with convex objective function, set $\mathcal{P}$ contain probability distributions defined on convex subset $\Omega \subset \mathbb{R}^n$ and with finite expectation value then we can say that the "worst" distribution is discrete and concentrate in at most $n + 1$ boundary points of Ω.

In general, theoretical analysis of the minimax stochastic problems is very difficult topic. Theorems and other theoretical results usually involve many conditions and cannot be generalized. But there can be absolutely different method of computing a solution and the method can be applied for very wide class of minimax stochastic models. The method is special heuristics - concretely genetic algorithms.

2 Heuristic methods

An optimization deals with finding extreme values for a solving problem. But for many real life problems it is very difficult to find an appropriate mathematical form which can be exactly solved by any of known and efficient algorithm. There are classes of problems which mathematicians study for tens or hundreds of years and still are not efficiently solvable. Due to the growing computational power of modern computers we can try to solve these problems by using "*brute force*". Many optimization tasks can be interpreted as searching for the best possible configuration of studied model. Thus for an exact solution we can just construct every possibilities one by one, compute their properties and after that choose the most appropriate one. Drawbacks of this approach are obvious. Let's consider the most famous problem of combinatorial optimization, *The Travelling Salesman Problem* (TSP).

For a given number of cities and given costs of travelling between each two of them we need to find the cheapest way with visiting each city exactly once and get back to the starting point. In other words, the TSP problem deals with searching for the combination of cities with the cheapest total cost. By using the brute force approach we need to explore number of possibilities which is increased with a growing number of cities. Exactly for TSP problem with n cities and given starting point we need to compute costs for $(n-1)!$ possible ways. By considering symmetry we can reduce the number of searching space to $1/2(n-1)!$ possibilities, but just for 60 cities it gives about 10^{80} combinations and for 100 cities it is even about 10^{160} combinations and that is practically impossible to explore with contemporary computers.

A requirement of solving the problems with rapidly increasing searching space leads to development of new methods which doesn't explore whole space but just its certain subset. There are many ways how to choose the searching subspace and some of them aren't based on well-developed mathematical theory but are inspired by intuition or observing a real life. These methods are called *heuristics*[4] and usually cannot guarantee obtaining the optimal solution for a modelled problem. Effectiveness of heuristic methods highly depends on a solved problem and adjustment method's coefficients. But if we know that a certain class of problems is solvable by a certain class of heuristic methods then we can dramatically reduce computing time and solve the problem in very efficient way. Many heuristic algorithms can guarantee non decreasing quality of the solutions obtained during the searching process, some algorithms can be successfully applied to very wide class of problems and other algorithms are designed just for specific problems. A choosing the most efficient algorithm for particular problem is a difficult topic and one should perform several tests before taking the final decision. A modern research in heuristic methods is very progressive and it is concentrate in a development of new methods, improving the known ones as well as investigation of theoretical and mathematical properties. By combining of some algorithms there can be introduce brand new one with some useful properties or by applying other algorithms in sequence we can iteratively improve the obtained solution. The algorithms can be described according to their strategy and a motivation of exploring the searching space.

[4] Due to Archimedes famous exclamation "Eureka!" (I've found it!).

2.1 Hill Climbing Algorithm

Very intuitive approach for finding extreme values is inspired by *hill climbing* [7]. Let's suppose we are interesting in finding maximal value of any discrete function and let's suppose we've randomly chosen one point from the search space. So the only what we need to do is to determine values of all points in a neighbourhood of our chosen point and move to the point with highest obtained value. This simple procedure is repeated as many times as there exists any higher value in the actual neighbourhood. As same as hill climber is able to find a top of a hill by this approach we are able to find a top of the objective function in some neighbourhood of our chosen point. In general, we cannot find the highest point in the whole searching space but just in some its subset. The methods which explore neighbourhoods are called *local search methods* and they are designed just for finding local extremes. Sometimes it is exactly what we need (e.g. if we just want to improve the candidate solution), but many times we need to know global function's extreme. If the objective function is not convex, i.e. has more than one peak, we can increase probability of finding the global extreme by repetition of the algorithms with different starting points. The same motivation can also be used for exploring continuous function, the method is called *gradient ascent method* and it doesn't determine values in neighbourhood (because there aren't no concrete neighbourhood points in continuous functions) but it deals with gradient of the function and moves in the direction with the highest slope. The hill climbing algorithm is, in fact, *deterministic*, the only random influence can occur just in choosing the initial point.

2.2 Tabu search

Classical local search methods, as hill climbing, moves from a solution to a neighbourhood solution when the objective function value of new solution is higher than the actual one. There can occur a problem how to deal with situation where the highest value of neighbourhood point is equal to the actual value. Let's suppose we design an algorithm to accept points with the same value and we move there. The problem arises when in the neighbourhood is not any higher value and therefore we have to move back to the initial point because the highest obtained value from the actual neighbourhood is right there. This causes that the algorithm never stops because we will all the time move between two point in a cycle. To design an algorithm without acceptation points with same value is not beneficial method because there cannot arise a cycle and therefore we cannot be able to find the best possible solution. One simple approach, how to deal with the problems like this, is called *tabu search* and it is based on in maintenance a list with a few last visited points [13]. So the algorithm explores neighbourhood of the actual point and checks whether the point, where it wants to move, is in the tabu list or not. If the new point is in the tabu-list the algorithm omits this point and checks another one. Thus the algorithm moves to the new point just in case that this point is not appeared in the tabu-list. If the new point is accepted the tabu list is updated by the last visited point. This prevents the method from doing an infinite cycles. But it is impractical to maintain list of all visited points, because the checking whether a point is contained there can take a lot of computing time and so we usually maintain list with just a few last points. The number of points in the tabu-list can hardly influence a behaviour of the algorithm. The more numbers in

the list the longer cycles we can avoid. Therefore we need to find a compromise between length of the tabu-list and supposed length of cycles. The tabu-list can also help in situations where in the neighbourhood are more equal highest values and we don't know which one to choose (therefore we can try one and if we come back we try the other one because of the tabu-list).

2.3 Simulated annealing

Optimization by *simulated annealing* is inspired by annealing metallurgy [15]. If any metal material is heated (because of forming, forging, etc.) and subsequently cooled off then atoms can get stuck in some inappropriate position and therefore there can arise some defects. For avoiding these defects there exist a method called annealing which controls decreasing temperature and therefore lets more time to atoms to find more appropriate configuration with a lower energy.

This method is from class of stochastic algorithm what means that there is some random influence and each run of the same algorithm can give different solutions. The randomness is contained in decision whether the actual point is moved or not. At the beginning we choose initial point and compute its objective function value, after that we choose some point from the neighbourhood, compute its value and decide if we move there or not. The decision is based on probability function where the main importance has so-called *temperature*. Before a start of the algorithm we choose a value of the initial temperature and during the process we decrease its value which simulates cooling in the annealing. Whole search space represents cooling metal and the moving point is an atom what search for place with the lowest energy. So after choosing point from the neighbourhood we decide according to the actual temperature value whether the new point is appropriate as a new position. For higher temperature there is higher probability of moving point. So on the beginning of the algorithm, when the temperature is the highest, the moving of the point is almost random and almost doesn't depend on objective function values. After the each iteration the temperature is decreased and the probability of the moving more depends on the values in the explored points than on the temperature. Before the finishing of the algorithm the decision whether to move depends just on the objective function value and therefore the point moves just to the position with better energy value. A rate of cooling is not constant and the faster solution converges the faster temperature decreases. The algorithm is in fact hill climbed algorithm with possibilities move to point with lower values. The main advantages are wider explored search space and possibility to avoid a stagnation in a local extreme.

2.4 Ant colony optimization

Many inspirations come from nature. The nature is able to handle very difficult optimization problems and that's why many scientists observe its behaviour. By observing how ants search foods in anthill's surroundings we can introduce an interesting optimization approach [10]. At the beginning each ant leaves the anthill in a random direction and looks for food. As soon as it finds something it returns back to the colony. During the whole searching the ant lays down pheromone trail to know how to gets back. The strength of the pheromone gradually vanish and if the ant spends a long time by searching it can vanish completely. On the contrary if the ant finds a food

soon then during the going back it lays down further pheromones what means that the trail has much stronger smell. Other ants, which haven't found any food yet, still randomly pass through the anthill's surroundings and if they cross the pheromone trail they follow it because they can suppose any food on its end. The more ants follow the trail and more ants go back with some food, the more pheromones are on the trail and therefore more further ants will use this way for transport the food to the colony. So this is the approach how ants find the shortest way from an anthill to the closest food source. Hence in the ant colony optimization algorithm the search spaces is analogy for anthill's surroundings, several moving points represent the ants and the objective function values determine amount of the food in some place. Each point in the search space have a variable value determining the level of pheromone left by ants which passed through it. In each algorithm iteration the pheromone level either increases if any ant passes through the point or decreases otherwise. Great advantage of this algorithm is in real-time problems, where the search space is changed during the optimization. The ants are able to react to a changing environment very quickly and still give very good solutions.

2.5 Bees optimization

Another highly biologically inspired optimization algorithm is based on behaviour of a bees colony [16]. When a colony of honey bees look for food (pollen or nectar) they can spread over long distance[5] in many different directions and due to their inherent algorithms they are able to concentrate on the places with more flowers whereas on the places with less food there are much less number of bees.

The optimization algorithm starts with letting the artificial bees out from the beehive in random directions, it means the algorithm generates certain number of bees in random places over the search space. Each bee explores its neighbourhood and remembers the place with the best objective function value which it has found. Each of the artificial bees also knows the place with the best value which has been discovered by any of the bee from the colony. Thus each bee remembers two places, the first one is its own maximum and the second one is maximum from the whole colony. In the real world the sharing of information about colony maximum is done by some kind of "dance". When the bees with a food come back to beehive they start to dance and the others bees are able to recognize where is the food source and how big it is. The bees further explore the search space in a direction compounds from the own and from the colony maximum and therefore they can concentrate on the places with the most of a nectar or a pollen.

2.6 Evolutionary Algorithms

The most known class of biologically inspired optimization algorithms is so-called *evolutionary algorithms*. This approach is based on theory of evolution discovered in the 19th century by French scientist Lamarck[6] who tried to explain the animal's adaptation to their environment as inheritance

[5] A honey bees can extend itself up to 14km from their beehive

[6] Jean-Baptiste Pierre Antoine de Monet, Chevalier de Lamarck (*August 1, 1744 – †December 18, 1829)

features which the parents used the most. For instance, if some animal use certain muscles then they become stronger whereas other muscles which are not very used become weaker. Thus the offspring of the parents with certain stronger muscles has the same part stronger also and the nonuse parts gradually atrophy.

Another approach for describing an evolution was introduced by English scientist Darwin[7] who described adaptation as a consequence of natural selection. The theory of the natural selection claims that if there are more individuals in the environment with limited food sources, then there arise a competition among the individuals and only the best ones can survive. Therefore if there exist any inheritable variation which helps to the individual be faster, stronger, etc. then the individual is more likely to survive and have a child with the same variation.

Neither Lamarck's nor Darwin's theory weren't considered to be sufficiently accurate and they cannot explain every variations and adaptations which were observed in the nature. The missing link was brought by Austrian monk and scientist Gregor Mendel[8] by his famous experiments with genetics of plants. According to his laws of inheritance every inheritable feature is coded in *chromosomes*. Each chromosome contains *genes* which are compound from two *alleles*. These laws are designed for sexual reproduction and during the crossover a new offspring gets from each parent one allele for each gene. The new chromosome is compound from genes which can be either homozygous (both alleles are same) or heterozygous (the alleles are different). A *phenotype* is a real property of an organism which is coded by the certain genes (for example, colour of eyes, height, ...). Therefore the phenotype of the new gene is determined by the two alleles inherited from the parent's genes. There are three types of alleles: dominant, recessive and codominant. The recessive allele cannot influence the phenotype if it is in pair with a dominant one, the recessive phenotype arise only when both new alleles are recessive. If the two dominant alleles are together in a new gene and the new phenotype is different from the phenotypes of each dominant allele then we speak about codominat alleles.

The Figure 2.1 shows so-called Punnett[9] square which describes four possibilities of crossover flowers with given genes. Both flowers has a gene which determines a colour of their blooms in the form *Bb*. This means that gene has two alleles, the B is dominant and determines red colour of the bloom and the b allele is recessive and determines violet blooms. So all combinations with dominant allele B have red blooms and just one possibility where two recessive alleles b appear has violet colour.

For surviving of species the inherited variance and mutations have to be appropriate for current environment. By Darwin's theory this requirement can be kept when the parents for crossover are chosen according to their abilities. It means that just the most adapted individuals can find a partner for crossover. There exist many different kinds of parter selection, e.g. one male with many females or faithful pairs of parents. The selection depends on a lifestyle of the species, some animals live in colonies others live alone and they meet possible partners very rarely. Theory of selection is very important for evolution and for keeping gene diversity in the population. The populations

[7] Charles Robert Darwin (*February 12, 1809 – †April 19, 1882)
[8] Gregor Johann Mendel (* July 20, 1822 – †January 6, 1884)
[9] British geneticist Reginald Crundall Punnett (*June 20, 1875 – †January 3, 1967)

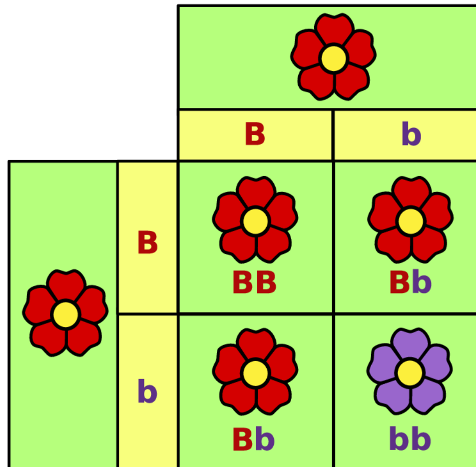


Figure 2.1 Punnett square describing dominant and recessive alleles

with small gene diversity can be very well adapted for actual environment but if something is changed (temperature, food sources, enemies, ...) then the species cannot be able to survive.

2.6.1 Genetic algorithms

The optimization algorithm inspired by the theory of evolution and genetic biology have become popular in the early 1970s when John Holland[10] has published his book where defined a framework for using evolution as an optimization method.

The main idea of genetic algorithms is to encode all possible solutions into chromosomes and by simulating crossovers and natural selections find the best solution [8]. More precisely genetic algorithms have an iterative character, as same as many others heuristic methods, and it means that the solution is not obtained just in one run of the algorithm but it is gradually enhanced during repetition of the same algorithms steps. And unfortunately, the genetic algorithms, as well as the theory of evolution, is too rich and too diversified to have a general description form which can describe all possible realizations. But there exists [2]a form which can describe the most usual types of genetic algorithms implementation.

Genetic algorithm is a stochastic heuristic algorithm contains following operators and parameters:

$$GA = (N, P, f, \Theta, \Omega, \Psi, \tau) \tag{2.1}$$

where P is a *population* which contains N individuals and because of the population is changed during the algorithm then the actual state in time t is defined by

$$P(t) = \{s_1, s_2, \ldots, s_N\}.$$

[10] John Henry Holland, American scientist and Professor of Psychology and Professor of Electrical Engineering and Computer Science (* 2 February 1929)

Each member (*individual*) s_i, for $i = 1, \ldots, N$ is sequence or set of real numbers (also very often integer numbers) of fixed length n which represent a solution of the problem. Therefore we can write $s_i \in S \subseteq \mathbb{R}^n$. Where S is set of all possible individuals.

f is a *fitness function* which maps individuals to real numbers $f : S \to \mathbb{R}$

Θ is *parent selection operator* which select u individuals from population P

$$\Theta : P \to \{P_1, P_2, \ldots, P_u\}$$

Ω is set of genetic operators includes *crossover operator* Ω_c, *mutation operator* Ω_m and another operators dependent on solving problem or concrete implementation. All these operators generate v new individuals (*offspring*) from the set of selected parents.

$$\Omega = \{\Omega_c, \Omega_m, \ldots, \Omega_{\text{optional}}\} : \{P_1, P_2, \ldots, P_u\} \to \{O_1, O_2, \ldots, O_v\}.$$

Ψ is *deletion operator* which removes r selected individuals from the actual population and after that the r new offspring are added to keep the size of the population constant. Therefore

$$P(t+1) = P(t) \setminus \Psi(P(t)) \cup \{O_1, O_2, \ldots, O_r\}.$$

τ is *terminate criterion* which maps set of all populations to the decision variable which says whether the actual population meets the target of the optimization and therefore whether it should be terminated

$$\tau : \{P\} \to \{\text{true}, \text{false}\}.$$

The parent selection operator Θ and the set of operators Ω are usually stochastic whereas the reduction operators Ψ is usually deterministic.

The fundamental idea of GA is existence of *chromosome*. The conception of the chromosome is not strictly defined and can be very various. Each individual contains at least one chromosome which defined qualitative properties of the represented solution and the individual can also contains any further information dependent on algorithm implementation. The chromosome is compound of *genes* which are in computer implementation usually real or integer numbers and therefore the chromosome can be considered as a vector of the genes. In more general implementation the chromosomes can be also graph or any other theoretical structure.

If we consider an optimization problem of finding the global minimum of objective function f

$$\bar{x}_{\text{opt}} \in \underset{\bar{x} \in S}{\operatorname{argmin}} \{f(\bar{x})\}, \tag{2.2}$$

than we can formulate that the goal of the genetic algorithm is to find the closest chromosome from the set of all possible chromosomes S to the one which represents the optimal value $\bar{x}_{\text{opt}}$.

The stochastic character of crossover and mutation operators are realized by probabilities P_c, P_m. The application of mutation operator on certain chromosome $\bar{x}$ has to be realized in such a way

$$\lim_{P_m \to 0} \Omega_m(\bar{x}) = \bar{x}.$$

The probability of crossover P_c gives for each chromosome (in general, for each individual in the population) its probability of selecting by operator Θ. The value P_c is therefore for each chromosome different and it depends on a fitness function value. More precisely, the selecting probability depends on individual's contribution to improving whole population. This contribution (so-called *fitness*) is non-negative real number determined by mapping function F

$$F : P \to \mathbb{R}^+$$

and it has to satisfies

$$f(\bar{x}_1) \leq f(\bar{x}_2) \Rightarrow F(\bar{x}_1) \geq F(\bar{x}_2).$$

Of course, this implication is used only for minimizing optimization problem. The relation between values of the objective function and the fitness function are usually linear but it is not necessary condition and the relation can be any arbitrary general function. We can formulate the *normalized contribution* as

$$F'(\bar{x}) = \frac{F(\bar{x})}{\sum_{\tilde{x} \in P} F(\tilde{x})},$$

and it is obvious that

$$0 \leq F'(\bar{x}) \leq 1 \qquad \forall \bar{x} \in P,$$

$$\sum_{\bar{x} \in P} F'(\bar{x}) = 1.$$

The normalized contribution is very often interpreted as the selection probability for a crossover each chromosome

$$P_c(\bar{x}) = F'(\bar{x}).$$

Therefore this value gives a measure for representation certain genetic information in the next generation (in the next iteration step of the genetic algorithm). An application of the P_c value is realized by implementation of selection operator Θ.

A basic scheme of the genetic algorithm can be following:

```
1   Generating initial population (usually randomly).
2   Computing the fitness values for each chromosome.
3   Parents selection and generating new offspring.
4   Computing the fitness function for the new offspring and create a
    new generation with them.
5   Mutation and new evaluating mutated individuals.
6   If the termination criteria is not satisfied than repeat the
    algorithm since the step 3.
7   As the result of the algorithm is considered the individual with
    the best fitness value.
```

This scheme is just a basic idea of the genetic algorithms. Many different implementation can appear in a real applications and some certain steps can be omitted, added or modified. There is no exact framework for the genetic algorithms but the main inspiration (selecting parents, crossover, mutation) is usually the same and real applications vary in some specific implementation details.

2.7 Various implementations of genetic algorithms

When we decide to apply genetic algorithms for solving any optimization problem we need to deal with many implementations details. For particular problem there can be many possibilities which can solve it but there can be also many implementations which are not very useful for the problem and even can cause that we are not able to find appropriate solution at all. There are no general rules which can say whether to use some specific implementation for a specific problem. For creating a concrete algorithm which can solve the given problem we have to prepare set of test for deciding how the algorithm should be implemented. There are a lot of various possibilities which we should take into account.

2.7.1 Representation of chromosomes

The chromosomes are very fundamental part of genetic algorithms and because of many important parts are related to them it is very useful to focus on their computer representations. There can be many different ways how to encoded the solutions into chromosome representation and because of the different representations have the different properties we should choose the one which is the most appropriate for our problem and for our computing demands. The main differences between various classes of evolutionary algorithms are mostly given by representations of the chromosomes. Usually as a genetic algorithm is called such a class of evolutionary algorithms which uses binary coded chromosomes.

Binary chromosome is usually represented as a vector of binary variables (i.e. the variable which can have either value 0 or 1). An encoding a solution into binary variables can cause a few problems. Clearly the easiest encoding is for problems where we need to represent some decision variables. Well known example of the decision problem is so called *Knapsack problem* where we

know the total weight of the knapsack and for given set of items with known weights we have to decide whether the item is in the knapsack or not.

A little bit more difficult encoding is for representing integer numbers. The usual transformation of the numbers into binary based system has disadvantage which is called *Hamming barrier*. If we consider a seeking for an optimal solution which is equal to decimal numbers $64_{10} = 01000000_2$ and if the population contains just chromosomes represented number $63_{10} = 00111111_2$, we can see that their binary representations have almost all genes different and therefore it is almost impossible to change the value into the optimal one by classical genetic operators. This problem is solved by so-called Gray code which can encoded every two integer numbers n and $n + 1$ into binary sequences which differs exactly in one bit. For encoding real numbers into binary chromosomes can be used a way where we choose accuracy of the sought solution and transform the real value problem into a integer one. For instance, if we look for a real number with two decimal places precision we can use for encoding integer numbers and their values divided by 100.

There can be some encoding which can extend the properties of the chromosomes. An interesting extension is so-called *shades coding* and it deals with multiploid chromosomes. In fact, multiploid chromosome means that there is more than one value for the one gene. Each gene has "inside" a few values and they determine the "outside" value for the gene. For instance a gene with values (1,1,0) inside has the outside value equal to 1 and a gen with (0,0,0) behaves as gene with value 0. The shade extension means that there is not strictly given which internal combinations has outside value 1 and which have 0, for a few combinations of inside values is the outside given randomly and therefore the chromosomes have more variabilities and adaptabilities. The set of genes with random outcome is called a *shadow zone*.

Another way of encoding the chromosomes is to represent numbers by some relevant type in computer. It means that if we operate with integer numbers we can use an integer variable as a gene. This let us to avoid the demanding encoding and every computations with the chromosomes are easier. Encoding with non-binary variable is not called genetic algorithm but they are general evolutionary algorithm. This kind of algorithms behave a little bit different because the genes in chromosomes created by classical crossover do not change their concrete values. For similar properties as the regular genetic algorithms there should be designed special classes of operators.

2.7.2 Size of populations

The important parameter of genetic algorithm is size of a population. The number of chromosomes can very dramatically influence the behaviour of the algorithm. If we choose the number very high then the population have a few advantages. With higher probability there can be chromosomes which produce the desired solution and whole population has higher diversity. The high diversity means that the population can avoid to stuck in some local extreme and also it can have higher adaptability for non-regular searching space. A drawback of the high number of chromosomes is the time needed to process it. Obviously the bigger size of a population the longer time is needed for evaluating all chromosomes and it can cause a big trouble in case with very time demanding fitness function. The high diversity in the population also cause that the algorithm

have to converge much slower and therefore an obtaining a solutions is more complicated. So for choosing an appropriate size of a population we have to consider the computation resources and also we need to have some idea how the objective function looks like. The more complicated the function is the more population diversity is needed. There is verified by experiments that for most of real problems it is sufficient to choose the size of population between 50 a 200 chromosomes.

2.7.3 Initial population

A run of the genetic algorithm starts with initialization a population. This initial population is the first approximation of the solution and the algorithm gets first information about the searching space. Mostly there are used two main ways how to generate the initial population. The first approach is to generate it completely randomly and the second one is to use some already known solutions which have been obtained by another heuristic methods. The random approach can cover most of the searching space but we can risk to get inappropriate genetic information which cannot be crossover to optimal solution. On the other side if we don't have any useful informations about the characteristic of a searching space it is the only way how to deal with it and moreover genetic algorithms have been designed exactly for that kinds of problems. Sometimes it can take a lot of generations before the algorithm finds any quality genetic material and therefore it is very useful to combine both approaches and put into randomly generated population a few known and quality chromosomes. To generate a population in regular way, i.e. the chromosomes would cover a space in the same distances, can be dangerous because the regularity in the chromosomes can coincide with regularity in the searching space and therefore we would lose a lot from diversity.

2.7.4 Fitness function

The fitness function can evaluate all chromosomes and assign them relevant values. The easiest way of defining this function is to use an objective function of the dealing problem. Sometimes, an implementation of the algorithm, involves modifying the objective function, for instance if the used computer framework is programmed only for searching minima or if it is able to work just with normalized function. The fitness function can depend on used operators like crossover or mutation because theses operators can produce invalid chromosomes which cannot be able to decode or it is not possible to use in the model. Usually if we deal with creating invalid chromosomes we have three fundamental possibilities how to solve the problems. At first we can erase the invalid chromosome and try to generate new one as long as it is not a valid one, another possibility is to transform it into its valid equivalent and the last main possibility is to use fitness function to artificially decrease its fitness value and therefore increasing a probability of automatically erasing in some further generation. It can be sometimes useful to keep the invalids in a population because there is some possibility that by crossover two invalid chromosomes can be arisen the optimal one.

2.7.5 Parent selection

Evolutionists says that the probability of crossover two individuals increase with their abilities. In other words the better the individual is the higher probability it has for crossover. The same

principle is also the main idea of a parent selection in genetic algorithms. When an algorithm selects a pair of parents it mostly looks at their fitnesses or ordering based on them. The important role also plays a randomness, because there would be high tendency to degenerating a population without this influence. So among fundamental selections belongs these strategies

- *Ranking selection* is used for sequences of chromosomes ordered according to their fitness function values and parents are selected randomly in such a way that the highest probability have the individual with highest fitness function. In fact, the probability of selecting ith chromosome form the population of size N is given by

$$P_c(S_i) = \frac{2i}{N(N+1)}, \quad i = 1, \dots, N$$

- *Tournament selection* is inspirited by selection in the nature. If some males want the same female they usually have to fight because of proving their qualities. The same principle is used for selecting the partners for crossover in this strategy. There are randomly chosen a few individuals from the whole population and then the one of them which has the highest fitness is considered for a crossover.

- *Proportional selection* is a selection based on normalized fitness function and therefore the chromosomes can have value $F_n \in [0, 1]$. The value of F_n can be interpreted as probability of selection P_c and therefore it is easy to generate random numbers from interval $[0, 1]$ and by them to decide which chromosomes should be crossovered.

2.7.6 Crossover

A crossover is used for creating new individuals from the parents chromosomes with preservation certain parts of their genetic information. Usually the chromosomes have fixed length and therefore crossover can be performed as a selecting corresponding gene from some of the parents. The most used crossover are

- *One-point crossover* means that we find out a number of genes N in the chromosome and then randomly choose integer number n less than the length N and the new chromosome is created as a composition of the first n genes from one parent and the last $(N - n)$ genes from the second one. This can keep an important genetic information and at the same time it provides variability needed to successful run of the algorithm.

- *Multi-point crossover* provides more variability because there are chosen more divided points and therefore it can more changed the informations contained in the sequences of genes.

- *Uniform crossover* is the most variable operator because it can provide arbitrary combinations of the parent's genes. Each new gene has the same probability for obtaining the information from either first or second parent. In fact the uniform crossover is limit case of a multi-point crossover.

These crossovers cannot be used just for binary chromosomes but they are useful for each chromosomes with fixed length. The chosen crossover highly influence performance of the algorithm because it is main part of the approach of an investigating the search space.

2.7.7 Mutation

A mutation is one of the most important operators in genetic algorithms. During each generation we choose a few chromosomes and we change them some of their genes therefore the mutations is logical complement to searching strategy. Crossovers cause an investigating of solutions which has been arisen as certain combinations of already known solutions and on the contrary a mutation gives us an instrument how to investigate solution which are not able to get as a result of crossovers. Mostly a mutation is perform by random change of given number of genes in selected chromosomes. There are a lot another approaches which are used for some specials chromosomes, for instance for binary chromosomes is possible apply so-called inversion mutation which invert value (from 0 to 1 and vice versa) in selected genes.

The number of selected chromosomes for mutation is usually very small (about 2%of population size) because an algorithm with high number of mutants loses its evolutionary properties and becomes a random heuristic. But moreover the properties of mutations can be adapted for an evolution of a fitness function during an algorithm run, what means if the population does't have any improving during the last few generation then the number of selected chromosomes or mutation ratio is going to be increased because of obtaining new genetic material for successful running of the algorithm.

2.7.8 Integration offsprings into a population

After a creating new offsprings we have to some how decide which original chromosomes would be better to replace by the newly created ones. Mostly used strategy says that the new population would be composed from the best chromosomes of the original population and also of the best from new offsprings. Therefore both sets of chromosomes are ordered into a one and just first N individuals are considered as a new generation, the rest of the set is erased. If a set of offsprings is as large as whole populations then as an integration strategy can be used a replacement of all original chromosomes by the new offsprings.

Integration have a significant influence for determining a direction of a searching through the feasible space because it decides which chromosomes are used as a parents and which one are doomed. The high ratio of new chromosomes in a population can be dangerous for keeping an important genetic information because it is often very useful to have some certain chromosome in the population for longer time and therefore increase its probability for crossover with the right individual. And on the other side the low ratio causes low modification in genetic material in the population and therefore the algorithm can tend to stuck in a local extreme.

2.7.9 Termination criteria

If we study a problem and we have any presumptions about the desired solution we can construct a termination criteria for satisfying them. If we know approximated value of the solution or if we just want to reach some given value then the decision when the algorithm should stop is obvious. The more difficult situation is when we don't have any information and therefore we don't know whether the algorithm has found the global optimum. Mostly used criteria are based on number of generations thus we let the algorithm works for e.g. 30, 100 or 200 generations and then we stop it without any respect to behaviour of the algorithm in the final moment. Another approaches are based on investigating variations among the chromosomes or on a rate of evolution the fitness function.

There are no general rule which can decide which operators and criteria we should apply. The important thing about the implementation of genetic algorithm is at the beginning choose a robust implementation and after obtaining some insight to the behaviour of the problem we can try to apply more specific modification and set their constants to an appropriate values. Each heuristic method involves a lot of time of testing but there exists more than one implementation which can solve particular problem. Thus the time spent by looking for a functional heuristic can give us very beneficial satisfaction.

3 Algorithms for determining boundaries of stochastic programming models

Problems in stochastic programming are traditionally solve by an exact analytical approach. However using this traditional approach involves many efforts and many highly theoretical analysis. If the problem contains some non-linearity or if the feasible set is not convex then it becomes almost unsolvable by the analytical way. Therefore there is a place for using alternative methods which can help us with finding the solution almost without theoretical investigating. These methods can be aforementioned heuristic methods which are able to solve very large class of problems but they cannot assure successful finding of the optimal solution. That is why, for every particular problem is needed to prepare a set of tests which tries how quality solutions is the method able to gives.

3.1 Classical genetic algorithm

We deal with a problem of determining boundaries in a stochastic programming model. There can be used several heuristic methods and the most straightforward way is to use the genetic algorithms. Therefore problem of finding a distribution of probability with the worst or with the best values of objective function can be interpreted as finding a set of samples which the distribution approximate. In the terms of genetic algorithms we can consider one chromosome to be a set of particular samples of random variables from the studied model. Thus in the beginning of the algorithm there is generated population of chromosomes and their fitness function values are computed by solving approximated stochastic problem. For instance if we consider stochastic problem with random variables vector $\tilde{\xi}$

$$\min_{\boldsymbol{x}\in X}\left\{E_{\tilde{\xi}}(f(\boldsymbol{x},\tilde{\xi}))\right\}$$

we can define chromosome as

$$\text{chromosome}_i = \{\xi_{0,i},\xi_{1,i},\ldots,\xi_{n,i}\},$$

which represents n different observations of the random variable $\tilde{\xi}$ and its fitness function is given by

$$fitness(\text{chromosome}_i) = \min_{\boldsymbol{x}\in X}\left\{\frac{1}{n}\sum_{j=1}^{n} f(\boldsymbol{x},\xi_{j,i})\right\}.$$

After a first initialization of the population (usually random initialization) every chromosome is evaluated and than with standard rules of genetic algorithms we apply parents selecting, crossover and others genetic operators. And because there are selected a few chromosomes with the best fitness function values and they produce new chromosomes by applying a crossover then if the algorithm integrate into a population just the offspring with high fitness value then we should still get better and better solutions. The output of the algorithm is a set of chromosomes with the

best fitness function value therefore it is the set with either the highest or with the lowest objective function value in approximation of the stochastic problem.

This algorithm is very straightforward and can be easily applied into existing genetic algorithm framework. But it has a drawback in a number of necessary evaluating. In each generation there are N chromosomes and each of them represent n random samples. Thus after g generations we have to solve $g \cdot N$ approximations of stochastic optimization programs with n samples. In general, optimization problems are very time consuming, especially if they are not linear. A time needed for solving an optimization problem highly depend on size of the model, it means the higher number of samples n the more time is necessary for a computing. Unfortunately this relation is not linear but mostly is exponential what causes the problem unsolvable for large values of n.

3.2 An approach based on modified genetic algorithm

Another approach which allows decrease a number of solving approximated stochastic programs has been developed by Jan Roupec and Pavel Popela[11] in [1]. This algorithm is based on genetic algorithms but for obtaining a solution it doesn't exploit just chromosomes but it uses whole population. The main idea is that each chromosome represents exactly one realization of a random variable occurring in the model and whole population represent the approximation of the random variable distribution. Therefore the goal is to find the chromosomes such that the expression

$$\min_{x \in X} \left\{ \frac{1}{n} \sum_{i=0}^{n} f(\boldsymbol{x}, \text{chromosome}_i) \right\}$$

is minimal. This expression is just deterministic approximation of the stochastic optimization problem and can be called *fitness of the population*. The chromosome$_i$ is one realization of the random vector, therefore if the model contains, for instance, four random variables then

$$\text{chromosome}_i = \xi_i = \left\{ \xi_i^1, \xi_i^2, \xi_i^3, \xi_i^4 \right\}.$$

The main difference between classical genetic algorithm and this modified version is in computing fitness function for chromosomes. The classical computing fitness function is independent on the rest of population and if the value has been computed then it is fixed during whole process. In the modified version, the fitness function for each chromosome is dependent on the computed fitness value of the actual population and moreover the fitness of the chromosomes is varying in a time.

Therefore for evaluating chromosomes we need to compute

$$\boldsymbol{x}_{min} = \min_{x \in X} \left\{ \frac{1}{n} \sum_{i=0}^{n} f(\boldsymbol{x}, \text{chromosome}_i) \right\} \tag{3.1}$$

and for each chromosome compute its fitness value in the following form

[11] Both of them works at Brno University of Technology, Czech Republic.

$$fitness(\text{chromosome}_i) = f(\boldsymbol{x}_{min}, \text{chromosome}_i) = f(\boldsymbol{x}_{min}, \xi_i). \tag{3.2}$$

This formula comes from the opinion that if the population has low fitness then most of the chromosomes contained in it have the low fitness also. And similarly, if the population has high fitness value then the chromosomes should have the high fitness too. In fact it means that each chromosome is important part of the population and the better is the chromosome the better is whole population. Therefore we can suppose that if we select a set of the best chromosomes and crossover them thus next generation will contains chromosomes with better fitness value and therefore whole population will have better fitness value.

An scheme of the modified genetic algorithm is

```
1   Generate initial population.
2   Compute the fitness value for whole population.
3   By using population fitness value compute the fitness values for
    each chromosome.
4   Select parents according to their fitness values.
5   By applying crossover generate new offspring.
6   Evaluate new offspring and integrate them into the new generation.
7   Mutate selected individuals and compute their fitness function.
8   If the termination criteria is not satisfied than repeat the
    algorithm since the step 4.
9   The result of the algorithm is the set of all chromosomes
    represent distribution of random variables.
```

There are several ways how to evaluate new offspring and mutants. We can either use $\boldsymbol{x}_{min}$ which we have already computed from the population or we can compute new $\boldsymbol{x}_{min}$ from original population together with the new offspring. We will deal with this topic in the further chapters. In one generation we need to solve one or two approximated stochastic programs, it is at most $2 \cdot g$ (g is number of generations) computations and therefore this approach markedly decrease computations demands in comparison with classical genetic algorithm and allow us to deal with larger problems. Another important thing is a possible connection between the algorithm and some general optimization languages. If we have already prepared models in any of that languages, (like GAMS[12], AMPL[13]) we can just easily create a framework for the genetic algorithm and the evaluating of population and chromosomes can be yielded to relevant solver. This can save a lot of time and allow us to avoid possible errors caused by further remodelling.

[12] http://www.gams.com
[13] http://www.ampl.com

4 Testing the algorithm

The algorithm proposed in [1] and described in the chapter 3.2 is a heuristic algorithm and it means that we are not sure whether it's going to find the solution or not. It is designed on assumption that better populations have chromosomes with better fitness function. If we want to use this algorithm we should prepare a few tests to verify or disprove our ideas.

4.1 Models

For a verifying the algorithm we build several models and apply it to them. In other words, we try to find the probability distributions which cause the worst and the best objective function values for given model. For this purpose we establish following models.

4.1.1 Linear stochastic two-stage model

This model has been adopted from [1] and it describes melting control problem in the suitable furnace (cupola, induced, or eletric-arc). It is real model with real data and it is described by two-stage linear stochastic model with recourse

$$\left.\begin{array}{l} \boldsymbol{x}_{min} \in \operatorname*{argmin}_{x \in X} \left\{ \boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} + \sum_{s=1}^{S} p_s \cdot Q(\boldsymbol{x}, \xi^s) \right\} \\ \text{subject to:} \\ \qquad \boldsymbol{0} \leq \boldsymbol{x} \leq \boldsymbol{b} \\ Q(\boldsymbol{x}, \xi^s) = \min_{\boldsymbol{y}^s \in Y} \left\{ \boldsymbol{q}^{\mathrm{T}} \boldsymbol{y}^s \right\} \\ \text{subject to:} \\ \qquad \boldsymbol{l}_2 \leq T_1^s A_1 \boldsymbol{x} + A_2 \boldsymbol{y}^s \leq \boldsymbol{u}_2 \\ \qquad \boldsymbol{0} \leq \boldsymbol{y}^s, \quad s = 1, \ldots, S \end{array}\right\} \tag{4.1}$$

This model represents dependence between decision how many tons $\boldsymbol{x}$ of material is given into a furnace and a final cost impacted by random utilization. We need to produce given number of alloys (Iron, FeSi-1, Steel, ...) with given amount of charging material (carbon, manganese, silicon, chromium). After putting the material $\boldsymbol{x}$ into a furnace some random losses appear. These losses are defined for each charging material by uniform probability distributions. If the losses exceed given boundaries during the melting we need to compensate it by adding another alloys which are usually more expensive than materials from the first place. The goal of this model is to decide how many tons of the cheap materials should be melted to minimizing adding alloys in the second expensive stage.

The vector $\boldsymbol{x}$ represents first-stage decision for tonnage of materials (Iron, Spinput, FeSi-1, FeSi-2, Alloy-1, Alloy-2, Alloy-3, SiC, Steel-1, Steel-2, Steel-2) and

$$c^{\mathrm{T}} = (60.0, 129.0, 130.0, 122.0, 200.0, 260.0, 238.0, 160.0, 42.0, 40.0, 39.0)$$

is cost of each material. The second-stage decision, the tonnage of the alloys (A-C, A-Mn, A-Si, A-Cr), is represent by $\boldsymbol{y}^s$ (symbol s is used for denoting that the value is dependent on a random realization ξ^s) and

$$\boldsymbol{q}^{\mathrm{T}} = (800, 1500, 1900, 4000)$$

is its cost. The matrices

$$A_1 = \begin{pmatrix} 5 & 0 & 0 & 0 & 0 & 0 & 0 & 18.75 & 0.5 & 0.125 & 0.125 \\ 0.947 & 4.737 & 0 & 0 & 63.158 & 9.474 & 34.737 & 0 & 0.947 & 0.316 & 0.316 \\ 3.124 & 21.429 & 64.286 & 60.000 & 25.714 & 42.857 & 35.714 & 42.857 & 0 & 0 & 0 \\ 0 & 10 & 0 & 0 & 0 & 20 & 8 & 0 & 0 & 0 & 0 \end{pmatrix}$$

and

$$A_2 = \begin{pmatrix} 100 & 2 & 1 & 1 \\ 0 & 70 & 0 & 6 \\ 0 & 10 & 60 & 1 \\ 0 & 4 & 0 & 40 \end{pmatrix}$$

describe amount of charging materials in the alloys. The columns are representation for the input materials (Iron, Spinput, ...) and the rows are charging materials (A-C, A-Mn, A-Si, A-Cr). The boundaries for an amount of the first-stage input materials are determined by vector

$$\boldsymbol{b} = (\infty, \infty, \infty, \infty, \infty, \infty, \infty, 0.01, 0.10, 0.10, 0.10)$$

and for the second-stage there are the amounts unbounded (all bounderies are equal to infinity).

The random impacts are determined by diagonal matrix with coefficients τ_i with prescribed probability distributions.

$$T_1^s = T_1^s(\xi^s) = \begin{pmatrix} \tau_1^s & 0 & 0 & 0 \\ 0 & \tau_2^s & 0 & 0 \\ 0 & 0 & \tau_3^s & 0 \\ 0 & 0 & 0 & \tau_4^s \end{pmatrix}$$

The output alloys have have prescribed amount of charging materials given by lower boundary

$$\boldsymbol{l}_2 = (3, 1.35, 2.7, 0.3)$$

and by upper boundary

$$\boldsymbol{u}_2 = (3.5, 1.65, 3.0, 0.45).$$

There are three variations of the described linear stochastic model testing which we use for the algorithm. The differences among them are realized just in probability distributions and therefore all aforementioned data are valid for each of them.

Model I

The first model contains four random variables with uniform distribution

$$\begin{aligned} \tau_1 &= U(0.734, 0.866) \\ \tau_2 &= U(0.902, 0.998) \\ \tau_3 &= U(0.672, 0.733) \\ \tau_4 &= U(0.972, 1.000) \end{aligned}$$

and therefore the matrix T_1^s can have whichever combination of the values. This model is the most simple one because the variables are independent and they create a convex set (a four dimensional hyperrectangle).

Model II

In the second model we fix variables τ_3 and τ_4 for easier imagination and we set the first two variables according to the following formula

$$\begin{aligned} (\tau_1 - 0.8)^2 + (\tau_2 - 0.932)^2 &\leq 0.066^2 \\ \tau_3 &= 0.7 \\ \tau_4 &= 1.0 \end{aligned} \quad .$$

The set of possible realizations now create a circle with center in [0.8, 0.932] and radius equal 0.066. The set is still convex but the variables are not independent.

Model III

There are again fixed variables τ_3 and τ_4 in the third model and the non-fixed variables create a set in a shape of a "cross".

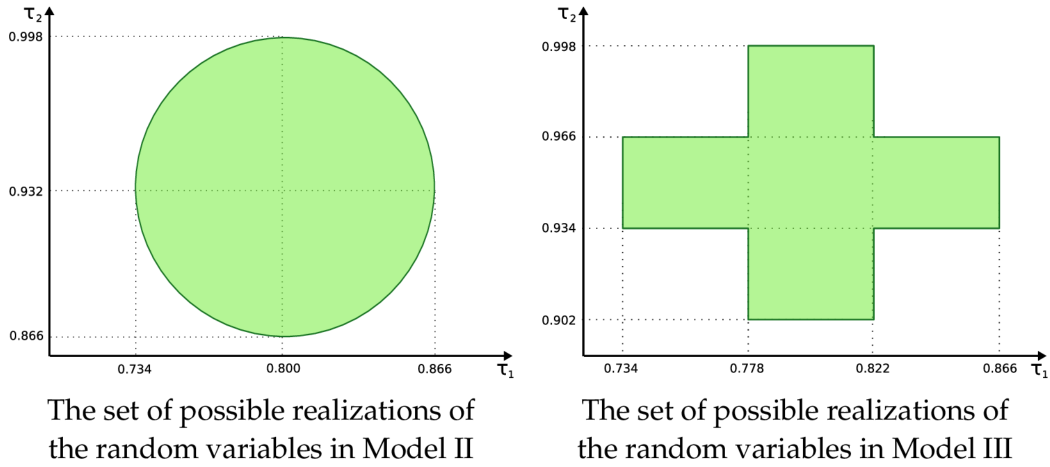
$$\begin{gathered} \tau_1 \in U(0.778, 0.822) \text{ and } \tau_2 \in U(0.902, 0.998) \\ \tau_1 \in U(0.734, 0.866) \text{ and } \tau_2 \in U(0.934, 0.966) \\ \tau_3 = 0.7 \\ \tau_4 = 1.0 \end{gathered}$$

4.1.2 Quadratic stochastic two-stage model

For more precisely testing we introduce a quadratic model with three random influences. This model is not based on real application, as the previous one, and it is imposes just for testing purposes. The model can be written in the following form

$$\min_{(x_1,x_2,x_3)\in X} \left\{ x_1^2 + 2x_2^2 + \frac{1}{2}x_3^2 + 2x_1 + x_2 - x_3 + \sum_{s=1}^{S} p_s Q(x_1, x_2, x_3, \xi^s) \right\},$$

where the recourse function is defined as



The set of possible realizations of the random variables in Model II

The set of possible realizations of the random variables in Model III

Figure 4.1

$$Q(x_1, x_2, x_3, \xi^s) = \min_{(y_1,y_2,y_3)\in Y} \left\{ y_1^2 + 2y_2^2 + 5y_3^2 + 3y_1 - y_2 + y_3 \right\}$$

with conditions

$$\begin{array}{rl} x_1 + \xi_1 x_2 + \xi_2 x_3 - 17y_1 - 5y_2 - 84y_3 & \leq -479 \\ \xi_3 x_1 + x_2 \qquad\qquad - y_1 + 2y_2 - y_3 & \leq -54 \end{array} .$$

and $x_1, x_2, x_3, y_1, y_2, y_3 \geq 0$.

The random influence is realized by three random variables ξ_i and according to their definition we can introduce next three models

Model IV

The set of possible realizations of the fourth model is a three dimensional hyperrectangle with uniform probability distributions such that

$$\begin{array}{l} \xi_1 \in U(-9.0, -3.0) \\ \xi_2 \in U(-13.0, -7.0) \\ \xi_3 \in U(-12.0, -2.0) . \end{array}$$

Thus the set is convex and all the variables are independent.

Model V

The Model V contains just two random variables (ξ_1 is fixed) and it is alternation for the Model II, therefore the set is a circle with center in the point [-10,-7] with radius equals to 3.0.

$$\begin{array}{c} \xi_1 = 6.0 \\ (\xi_2 + 10.0)^2 + (\xi_3 + 7.0)^2 \leq 3.0^2 \end{array} .$$

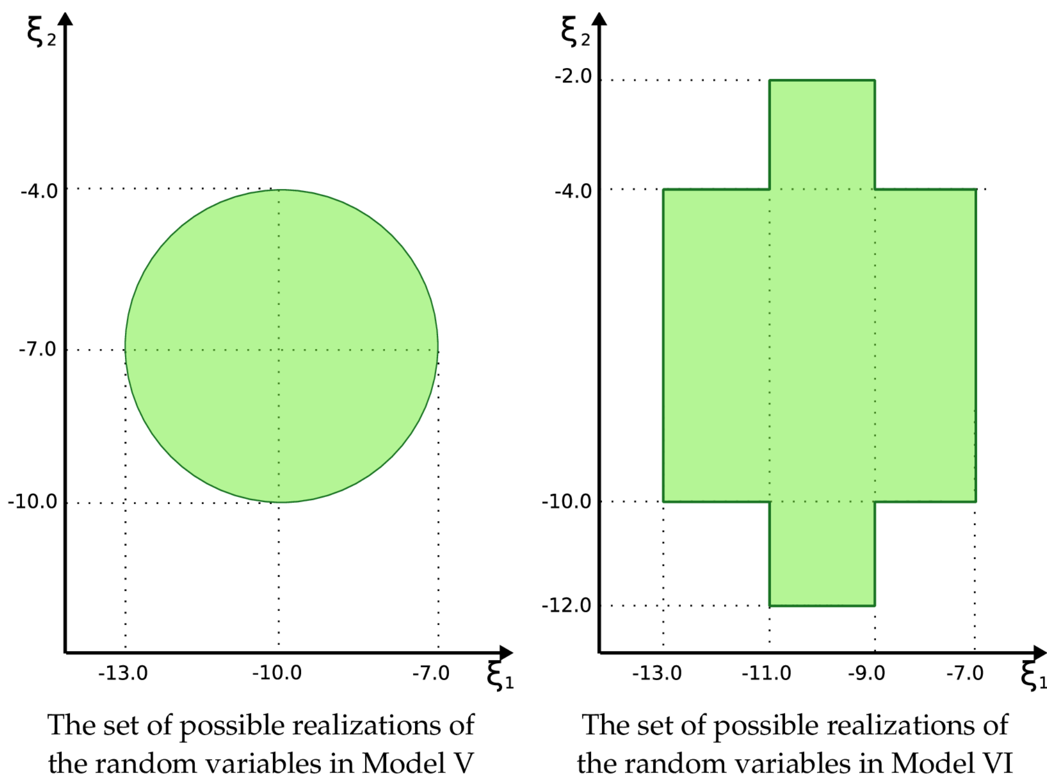
The set is convex but the variables are not independent.

Model VI

In the last model we introduce an alternative to the Model III and the set of possible realizations can be called as a "cross".

$$
\begin{aligned}
&\xi_1 = 6.0 \\
&\xi_2 \in (-13.0, -7.0) \text{ and } \xi_3 \in (-10.0, -4.0) \\
\text{or} \quad &\xi_2 \in (-11.0, -9.0) \text{ and } \xi_2 \in (-12.0, -2.0) \quad .
\end{aligned}
$$

This set is not convex and the random variables are not independent.



The set of possible realizations of the random variables in Model V

The set of possible realizations of the random variables in Model VI

Figure 4.2

4.2 Monte Carlo investigating of confidence interval

As we have seen in the chapter 1.3.1 there exists a Monte Carlo approach based on statistic methods how to determine a confidence interval for stochastic programming problems. Therefore for analysing the problem we have to find upper and lower bounds and determine the confidence interval (formula 1.15) by them.

For determining the upper bounds we need to generate n_u random samples, find some suboptimal solution $\hat{x}$ and by them compute objective function values for the each of the randomly generated sample. As an upper bound is subsequently used standard mean estimator defined in formula 1.13.

The lower bounds involve much more of computing power because according to the formula 1.14 there is computed n_l approximated optimization problems and their values are used for standard mean estimating. Each approximated optimization problem is compound from n randomly generated samples.

Clearly the confidence interval (formula 1.15) is given by these boundaries and we can just correct them by standard errors gained from the numbers n_l (respectively n_u) and desired confidence level $(1 - \alpha)$.

There is the table 4.1 with results of upper bounds computed for all models. There has been used suboptimal solutions $\hat{x}$ which have been computed as a result of approximated problem for 20 randomly generated scenarios. Thus suboptimal solutions for the linear models are

$$\begin{aligned}
\hat{x}_{\mathrm{I}} &= (0, 0.03080168, 0.02255713, 0, 0.02114842, 0, 0, 0.01, 0, 0, 0) \\
\hat{x}_{\mathrm{II}} &= (0, 0.03, 0.03452594, 0, 0.0220179, 0, 0, 0.01, 0, 0, 0) \\
\hat{x}_{\mathrm{III}} &= (0, 0.03, 0.03490856, 0, 0.02106134, 0, 0, 0.01, 0, 0, 0)
\end{aligned}$$

and for the quadratic models are

$$\begin{aligned}
\hat{x}_{\mathrm{IV}} &= (11.2215016, 9.39082217 \cdot 10^{-9}, 6.59765259) \\
\hat{x}_{\mathrm{V}} &= (8.09398534, 9.37849905 \cdot 10^{-9}, 6.95608609) \\
\hat{x}_{\mathrm{VI}} &= (8.47495486, 1.60872814 \cdot 10^{-8}, 6.77029895)
\end{aligned}$$

As a confidence level $1 - \alpha$ has been chosen level 0.995.

	$n_u = 10$			$n_u = 50$			$n_u = 100$		
model	$\bar{U}$	$\tilde{\varepsilon}_u$	$\bar{U} + \tilde{\varepsilon}_u$	$\bar{U}$	$\tilde{\varepsilon}_u$	$\bar{U} + \tilde{\varepsilon}_u$	$\bar{U}$	$\tilde{\varepsilon}_u$	$\bar{U} + \tilde{\varepsilon}_u$
I	41.232	6.294	**47.528**	42.386	2.589	**44.976**	41.880	1.873	**43.753**
II	37.148	0.053	**37.211**	37.171	0.026	**37.197**	37.172	0.016	**37.188**
III	37.170	0.032	**37.202**	37.165	0.014	**37.179**	37.155	0.011	**37.166**
IV	255.56	78.72	**345.08**	279.91	40.60	**320.51**	263.35	20.87	**284.22**
V	168.70	16.67	**185.38**	174.76	12.51	**187.27**	172.69	8.74	**181.43**
VI	229.88	153.26	**383.14**	200.15	33.20	**233.35**	212.79	28.45	**241.29**

Table 4.1

The estimates of the lower bounds for all models are in the table 4.2 and there has been used value $n = 20$ (n represents a number of generated samples in the approximated model). The confidence level is the same like for the upper bound estimating, therefore $(1 - \alpha) = 0.995$.

	$n_l = 10$			$n_l = 50$			$n_l = 100$		
model	$\bar{L}$	$\tilde{\varepsilon}_l$	$\bar{L} - \tilde{\varepsilon}_l$	$\bar{L}$	$\tilde{\varepsilon}_l$	$\bar{L} - \tilde{\varepsilon}_l$	$\bar{L}$	$\tilde{\varepsilon}_l$	$\bar{L} - \tilde{\varepsilon}_l$
I	41.990	1.398	**40.592**	41.929	0.387	**41.542**	42.163	0.391	**41.772**
II	37.148	0.024	**37.124**	37.152	0.013	**37.138**	37.150	0.008	**37.142**
III	37.025	0.032	**36.993**	37.018	0.012	**37.005**	37.028	0.007	**37.021**
IV	185.14	25.48	**159.66**	193.16	9.29	**183.87**	188.07	5.98	**182.08**
V	125.21	3.40	**121.81**	129.04	2.37	**126.67**	128.97	1.46	**127.50**
VI	165.42	23.56	**141.85**	160.78	7.80	**152.97**	158.32	5.10	**153.21**

Table 4.2

So the confidence interval for each model can be taken from these two tables in the form $[\bar{L} - \tilde{\varepsilon}_l, \bar{U} + \tilde{\varepsilon}_u]$.

4.3 Distribution of the fitness functions in the whole populations

The tested algorithm described in the chapter 3.2 used two kinds of fitness functions which are interconnected together. The more important one is the fitness function of the whole population, the algorithm is designed to find an extreme values of this function and the fitness functions of particular chromosomes are, more or less, just auxiliary. But if we deal with solving an srbitrary problem then it is usually very useful to investigate it as much as possible and therefore we have decided to analyse distribution of particular chromosome fitness in different populations.

For a graphical representation of all distributions we need to generate data which cover almost all possibilities in the searching space. We can either generate populations and chromosomes in regular distances or we can try to generate them absolutely randomly. The regular data have disadvantage that their pattern can appear in the results and therefore it may cause wrong conclusions.

So we have generated 30000 random populations with given fixed number of scenarios. For all of the populations we computed their fitness function as well as fitness functions for each of their chromosome. For the resultant chart we have to choose only the populations which have fitness function in some small interval a find out the distribution of all of their chromosomes. For an obtaining the probability distribution we need to normalised the data in such a way that the sum of all values is equal to one. This have to be done for each interval in a range of the population fitness function. We have executed this investigation for Model I and Model IV and for all numbers of scenarios between 1 and 29.

Model I

How to read the charts. The row with population fitness value shows a probability distribution of appearing certain chromosome fitness. For instance, the Figure 4.3 describe populations with two chromosomes, the row with a value equals to 40 says that the population with this fitness

can have chromosomes with fitness values in the interval [35,37] or in the [43,45] and all points of these intervals have approximately same probability equal to c.c.a. 0.1. The value of probability is expressed by a color intensity which is shown in the color bar on the right side.

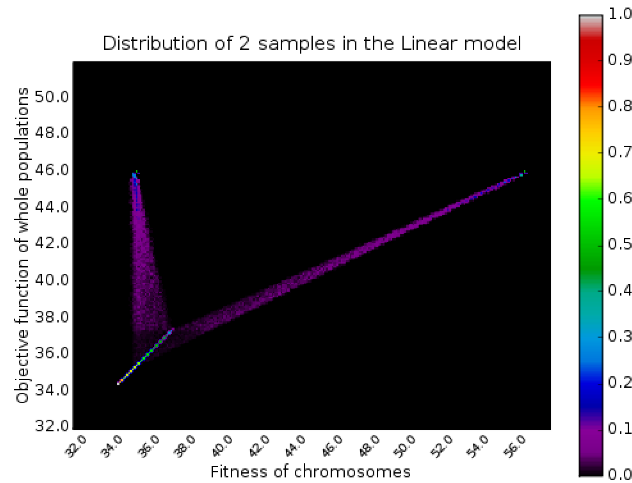


Figure 4.3 The distribution in the populations with two chromosomes

If we consider the approximated linear problem with two scenarios we can observe that the lowest population fitness value is in the case when the scenarios have the same fitnesses, therefore the two scenarios are in the some point. And on the contrary, the highest fitness of population comes when the scenarios have very different fitnesses, the first one has the highest possible value and the second one has the lowest one.

The Figure 4.4 shows probability distributions in populations contain 16 chromosomes and it says similar results like in the populations with two chromosomes. The populations with the low fitness have their chromosomes concentrated mostly in the part with lower chromosomes fitness. And the highest populations tend to have chromosomes divided into two parts, one with high and the second one with low fitness. Another examples are shown in the appendix A.

The populations with a higher number of chromosomes have very similar distributions and the higher number of scenarios the more fuzzy is the resultant chart. It means that the weight of the one particular chromosome is decreasing with increasing the size of population. In other words, one particular chromosome cannot change a fitness of a large population very much.

Model IV

The probability distributions in the quadratic Model IV are a bit complicated than the linear one. The Figure 4.5 shows populations with five chromosomes. The distributions change shape during

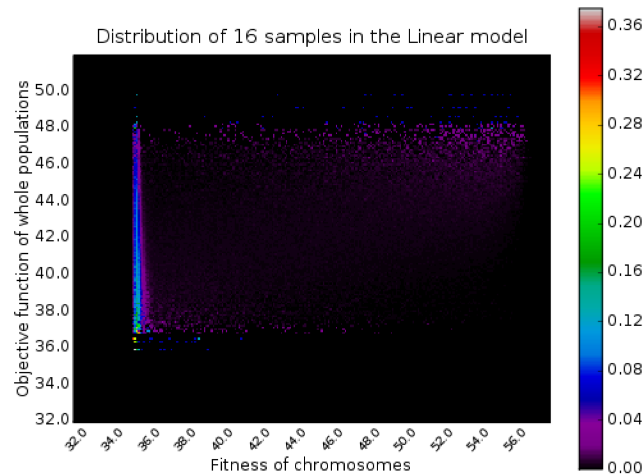


Figure 4.4 The distribution in the populations with 16 chromosomes

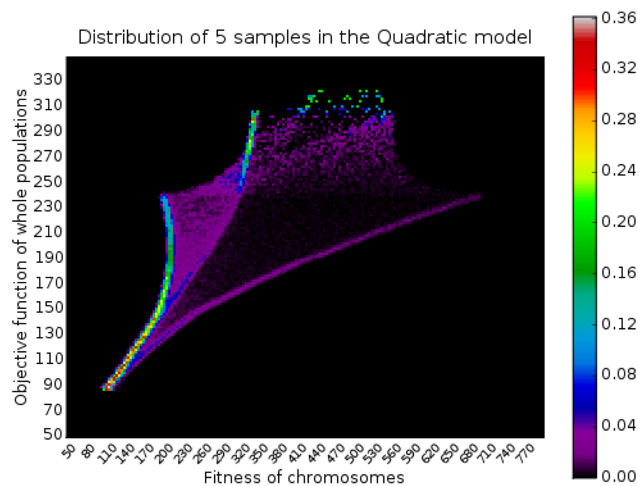


Figure 4.5 The distribution in the populations with 5 chromosomes

increasing population fitness and also very rapidly change the width of its support. But it is valid that the lesser number of low chromosome fitness the higher fitness the population has.

The Figure 4.6 shows distributions of population with 14 chromosomes and it is very similar for every charts with higher number of chromosomes in a population. The distributions preserve the shape of growing population fitness, what means that in the higher population fitness there are chromosomes with higher fitnesses also.

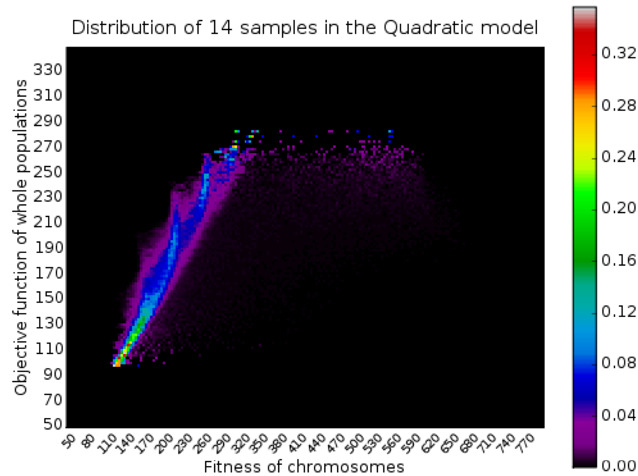


Figure 4.6 The distribution in the populations with 14 chromosomes

This analysis shows that there exist some dependence between population fitness and fitnesses of their chromosomes and therefore a using of the heuristic algorithms based on this property is reasonable and we can suppose to get some useful results.

4.4 Evaluating of populations

One of the main characteristics of the genetic algorithms is that each chromosome has its own fitness function value. The tested algorithm is based on the idea that we can compute $x_{\min}$ from the formula 3.1 and subsequently we can use this value for an evaluating each chromosome (formula 3.2). But there can arise a problem with evaluating chromosomes after the reproductions and mutations. Let's have a look at the situation when we have a population with n chromosomes and we have evaluated them, selected parents and created m new offsprings. Now we have at least two possibilities how to determine fitnesses for the new offsprings.

- We can use the $x_{\min}$ which we have computed from the population of the n original chromosomes
- or we can determine a new value $x_{\min}$ by computation it from the population of n chromosomes together with m new offsprings together.

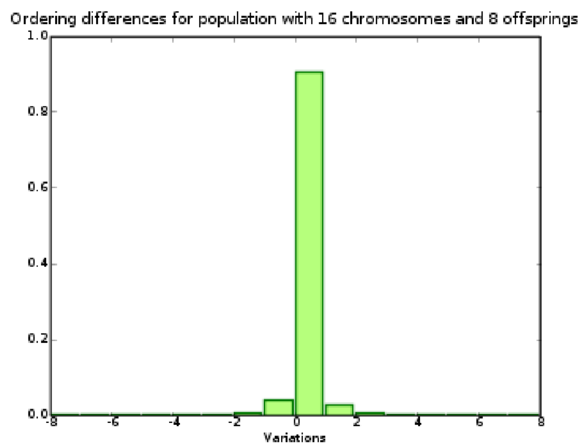
The main advantage of the first approach is a saving computations resources because to compute the solution of the problem with $n + m$ samples can take considerable amount of time. The main advantage of the second approach is an accuracy. The first approach can give values which are not correct because the $x_{\min}$ used for computation is determine for another set of scenarios and

therefore values based on it can mistake us. For taking the decision which approach should be used there has been prepared a test which can help us with it.

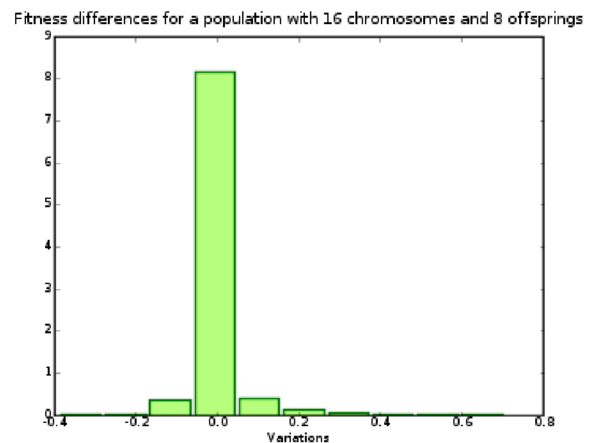
The test says us how much the fitness function values computed by these two approaches differ. So for the Model I (the linear model) and Model IV (the quadratic model) we've randomly generated 500 populations with n chromosomes and we have computed corresponding value $\boldsymbol{x}^0_{\min}$. By this computed value we have determined fitness function values for each of the chromosomes and also for m another randomly generated offsprings. Further step is to compute $\boldsymbol{x}^1_{\min}$ given by a population of all the chromosomes together (therefore from $n+m$ scenarios) and based on this new value we determine new fitness values for all them.

Thus we have two sets of fitness values related to the two investigated approaches. In the algorithm the computed fitness values are used for deciding which of the new offspring should be integrated into a proper population. And because of many integration strategies depend on the order of the chromosomes we should also investigate how much modified are the ordered populations. In other words, if the ordered set of chromosomes evaluated by the first approach is the same or similar as ordered set evaluated by the second approach.

We have tested all populations between 2 and 20 and we have compared differences in ordering and fitnesses with number of offsprings. The resultant data are shown in the appendix B. Some typical results are shown in the figure 4.7. The left figure shows a histogram of differences between ordering (it means $position_0 - position_1$) and the right one shows a histogram of differences between fitness (i.e. $fitness_0 - fitness_1$). The figures are computed for population with 16 chromosomes and 8 new offsprings in the Model I.



The differences in the order



The differences in the fitness

Figure 4.7

So after a consideration of all these facts we have decided to implement the first approach which use the old $\boldsymbol{x}^0_{\min}$ because it saves a lot of computing time and the caused inaccuracies are not so important, especially the orders are almost the same.

4.5 Computer implementation

For testing an algorithm we have created a computer script written in scripting language Python on operating system Linux. There are several ways for developing the algorithms. It is usually considered that the most intuitive and most extensible way for representing an algorithm is an object representation[14] and therefore we have chosen it also. The program have no graphical user interface and it is fully controlled by text files. A running of the program involves installed python interpret and its library for solving optimization problems OpenOpt [6].

There has been developed full evolutionary algorithm allows us easily extend and modify it. At the beginning, the whole algorithm has to be compose of the components which are appended in the modules. The chromosomes are represented as vectors of real numbers and all operators are designed just for them.

A calling of the program is realized by text file and there is shown a simple example

```
galg=CGeneticAlgorithm()

#definition of Chromosome
galg.setParameters(chromosome=COneScenario,
                   numberOfChromosomes=30)

#definition of reproduction
galg.setParameters(selectParents=bestWithOthers,
                   selectParentsParameter={'couples':15},
                   crossover=uniformCrossover,
                   integrateOffsprings=betterIntegration)

#definition of evaluation
galg.setParameters(evaluateChromosome=putX)
galg.setParameters(evaluatePopulation=evaluate)

#definition of comparing between two chromosomes
galg.setParameters(compare=minimize)

#definition of mutation
galg.setParameters(selectMutants=uniform,
                   selectMutantsParameter={'number':1},
                   mutate=changeNComp,
                   mutateParameter={'gens':2})

#definition of termination
galg.setParameters(terminate=generationTerminate,
```

[14] Object-oriented programming

```
                terminateParameter={'lastGeneration':30},
                terminateParameterFunction=generationTerminateParam)

galg.runAlgorithm()
print galg.getBest()
```

There is created an object `galg` which represents a genetic algorithm, it takes operators and their parameters by its method `.setParameters`. A running is executed by method `galg.runAlgorithm()` and we can get the computed results by another method `galg.getBest()`. Obviously there has to be imported a few modules with the operators, it is omitted here because of its unimportance. Let's remark the lines with `#` are comments and they are not needed for algorithm run.

The whole script is saved into a text file and easily executed from the command line `$python file.py`.

4.6 Algorithm for searching minimal distributions

The implementation of an algorithm which should find the distribution of chromosomes with minimal fitness value can be considered as genetic algorithm with typical operators. Textbooks and other resources usualy say that the typical implementation is based on

- The initial population is randomly generated.
- The selected parents have mostly high fitness value.
- The crossover is uniform.
- Chromosomes for mutation are selected randomly and uniformly
- Mutation changes selected genes by new ones randomly generated
- The offspring with high fitness are integrated into the population instead of chromosomes with low fitness value.

So this algorithm was implemented for Model I and its coefficients was set to some typical values (chromosomes in a population=30, number of offsprings=15, mutated chromosomes=1 and number of mutated genes=2). For an obtaining the results with lower random influence we ran the algorithm 30 times with 50 generations in one run.

Because this model is a testing model we know its true boundaries. The lowest value of the population fitness is approximately 34 and the highest one is about 55. The mean of the function values is almost 43. The figure 4.8 shows one typical evolution of the population fitness function with respect to the number of generation. The 90%of the algorithm's runs gave results in the interval (35.9; 36.8) so we can observe that the values found by the algorithm are relatively close to the lower bound (bottom red line) but, on the other hand, they should be closer. By the results we mean the best fitness population value obtained during the whole algorithm run and the 90%interval is useful because we can omit accidentally too good and too bad runs.

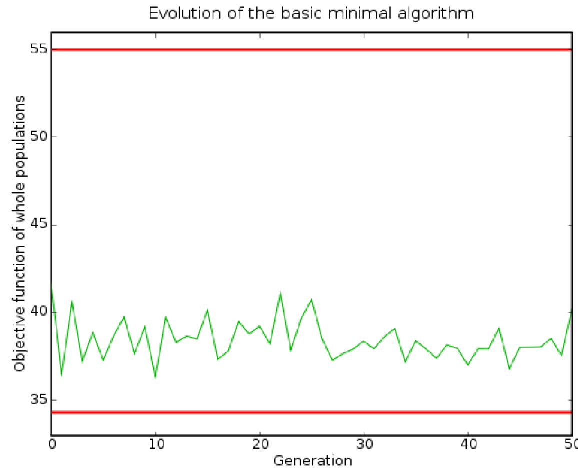


Figure 4.8 Typical run of the basic implementation.

Thus now we can try to enhance the algorithm by analysis and replacement some of the algorithm's components. We can investigate the influence of the components if we change them for another ones and observe differences in the outcomes. Clearly, it cannot be very precise because some components can coincidently collaborate and by removing one of them we change a behaviour of the another. But it can give us a first insight into the algorithm.

If we change the parent selection by random selection the outcome of the algorithms is almost not changed, so we can suppose we cannot deal with the selection a lot. On the other hand replacing an integration of offsprings into the population by the random one cause that the outcomes oscillate around the mean of the objective function and it means the the algorithm becomes practically random and it lost almost all advantages of evolutionary algorithms. In the same time, it means the integration of offsprings can be important for the run. The switching off the mutation implies that the outcomes are in the interval (36.7; 37.9) and therefore the mutation has positive influence for the results and by using it we are able to obtain better fitness function. Therefore if we would enhance the mutation then there exist a certain probability that the whole algorithm would be enhanced. What we expect from a mutation is supplying the population by new genetic information and moving the chromosomes to the new places.

We introduce new mutation operator, called *Touch*. Usual mutation replaces a few genes of the chromosomes by new randomly generated ones, the operator Touch just takes a chromosome and move it a little. Concretely, for our chromosomes with real numbers we just add a small number to each of them and therefore it causes that the whole chromosome is a little moved in the searching space. Our implementation involves one parameter *distance* which decides how long distance should be between the original and the new chromosome.

$$gene_{\text{new}} = gene_{\text{original}} + \text{random}(-1, 1) * distance, \quad \forall gene \in chromosome$$

This formula determines that the new chromosome lies in a sphere with radius *distance* and with center in $chromosome_{\text{original}}$. If we apply the operator to the chromosomes near a boundary of a searching space it can happen that the moved chromosome is outside the space, so an important part of the operator is checking whether it is located in the proper area and if not then correct it. Usually if operators of crossover or mutation can produce invalid chromosomes we have two way how to deal with it. Either we transform the invalid chromosome into the valid one or we let it invalid and just penalize its fitness value. An integration of invalid chromosomes into a population was considered as an inappropriate because the fitness functions of a population and each of its members partially depend on a position of all particular chromosome in a search space, therefore a computing fitnesses from a population with invalids can give wrong results and can negatively influence an algorithm's run. The repairing is implemented as a converting the invalid chromosome into the geometrically closest valid one.

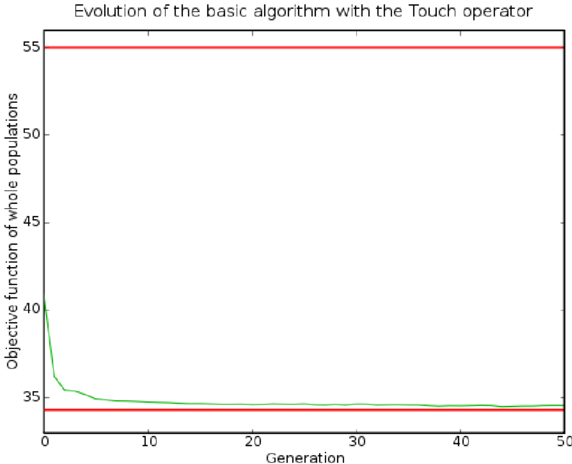


Figure 4.9 Typical run of the implementation with the operator Touch.

So by 30 runs of the basic implementation with the new mutation operator (the *distance* = 0.005 and 5 selected chromosomes in one generation) we obtain 90%of results in the interval (34.44; 34.66) and because of the real minimum is 34.3120 we can say that the new mutation changes the algorithm very noticeably and it works correctly for the Model I and moreover the sufficient results are obtained even after 10th generation.

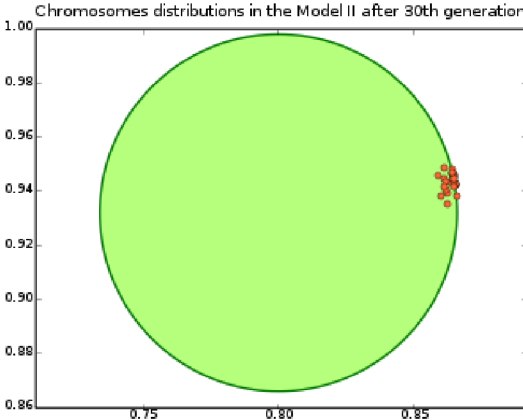
Now we demonstrate the results of the implementation the Touch operator for the next five models also.

	I	II	III	IV	V	IV
Real	34.312	36.807	36.741	77.762	87.846	82.303
Basic	(35.85; 37.06)	(36.93; 37.01)	(36.85; 36.89)	(116.40; 137.59)	(103.95; 113.73)	(106.05; 120.84)
Touch	(34.44; 34.77)	(36.75; 36.86)	(36.76; 36.78)	(77.77; 79.87)	(87.89; 89.71)	(82.85; 84.51)

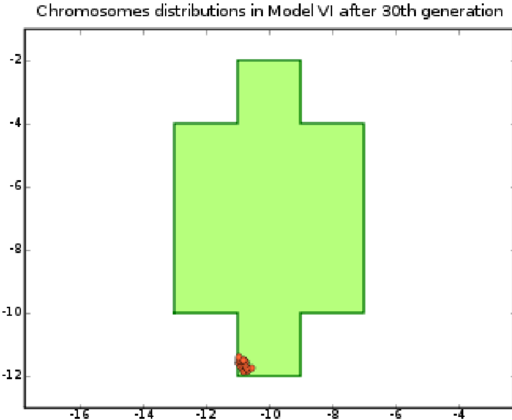
Table 4.3

The table 4.3 compares the algorithm with usual mutation and the one with the Touch mutation, for better comprehension in the first line there are written the real lower bounds of the models and the next two lines contain 90%intervals of results for particular implementations. We observe the Touch mutation gives results almost on the lower bound and especially, for the quadratic models give much better solution than the usual mutation.

It is also useful to know how the chromosomes move in the searching space during the run of the algorithm, so the figure 4.10 shows typical example how the distributions of 30 scenarios look like after 30 generations. We know that in the first generation the chromosomes randomly and uniformly cover whole searching space and after a few generation there is an apparent pattern of gathering chromosomes together and on the end of the algorithm they are almost in the one point.



The minimal distribution for the linear Model II



The minimal distribution
for the quadratic Model VI

Figure 4.10

Examples of evolution of fitness function and distributions of chromosomes for the rest of the models can be found in appendix C.1 and C.3.

4.7 Algorithm for searching the maximal distributions

By using the same algorithm with the Touch mutation to search for a distribution with maximal fitness function we obtain very disappointing results. By applying it to Model I we get distributions with value about 44 and it is very far away from the real maximum which is about 55. So we conclude the algorithm for searching minimal values is useless for searching the maximal ones. Therefore we have to change some operators and try to find better implementation.

The basic implementation, demonstrated in the section above, works much better but again, like in the previous implementation, it can give much closer results to the real boundaries. We know, from the stochastic programming theory, the distributions with highest cost are usually concentrated in several separated places. So we introduce new operator for integrating offspring into a population.

The classical integration is based on idea that the population should contain mostly chromosomes with the highest possible fitness but it is not useful in our implementation because the result of whole algorithm depends on all chromosomes and not just on the best one. Therefore we modify an offspring integration such that the chromosomes with average fitness are going to be removed and consequently the population will tend to separate into a few smaller crowds. This operator is called *Average Out* and its precise definition is following.

- Join a population together with new offsprings into one ordered sequence.
- Choose a final ratio between the chromosomes with high and low fitnesses.
- Create the population of the lowest and highest chromosomes with respect to the chosen ratio.

If we have n equal to a number of the original chromosomes then we can easily implement the ratio $r \in [0, 1]$ as preserving the first $\lfloor r * n \rfloor$ and the last $(n - \lfloor r * n \rfloor)$ chromosomes from the ordered sequence[15].

The chosen ratio for our implementation is $r = 0.9$ what means that the population contains 90%of low fitness chromosomes and 10%of the high ones. That is why we modify selecting parents operator to select almost all chromosomes with a high fitness. Therefore the new algorithm based on the basic implementation with modified parent selection, Average Out integration and Touch mutation gives results described by an evolution of the population fitness in the figure 4.11.

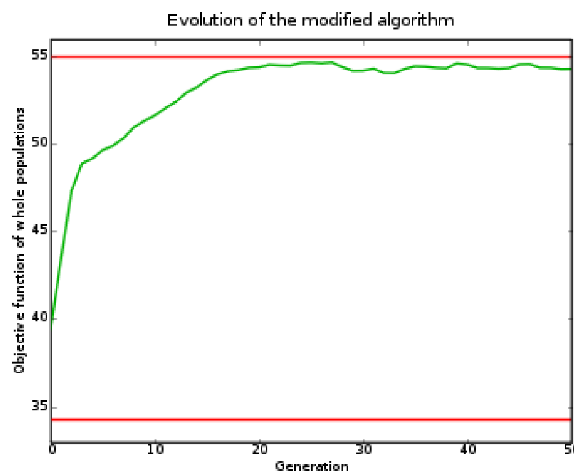


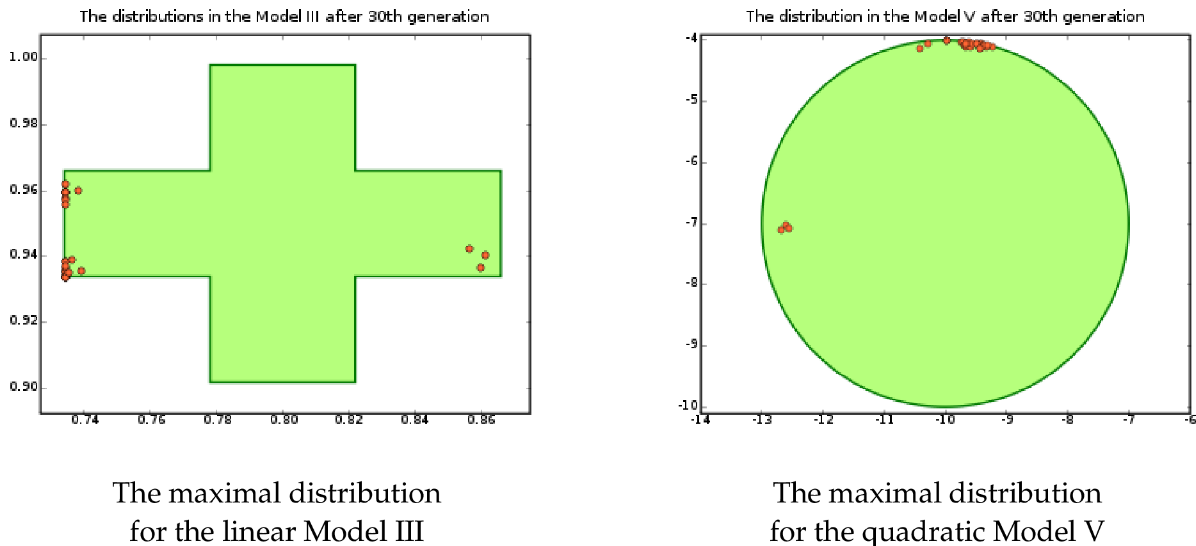
Figure 4.11 Typical run of modified implementation

For understanding the behavior of our new implementation we have applied it for all Models I-VI and recorded the 90%intervals into the table 4.4.

[15] the symbols $\lfloor x \rfloor$ denote the integer part of number x

	I	II	III	IV	V	IV
Real	55.41	37.32	37.14	360.17	161.82	359.51
Basic	(49.76; 52.48)	(37.21; 37.23)	(37.03; 37.11)	(236.75; 276.03)	(140.42; 146.98)	(183.34; 233.17)
Average Out	(50.46; 55.29)	(37.25; 37.29)	(37.10; 37.14)	(354.35; 356.26)	(150.93; 160.83)	(262.88; 353.84)

Table 4.4



The maximal distribution
for the linear Model III

The maximal distribution
for the quadratic Model V

Figure 4.12

Another examples of fitness function evolution and the chromosomes distributions for the rest of the models can be found in appendix D and D.3.

The results show an improvement against the algorithm with usual genetic operators, especially for the quadratic models is the improvement very significant. The figure 4.12 shows final distributions for the Model III and Model V and it can be seen the algorithm is able to find extremes in separate discrete points. For all of the models we are almost able to reach the real upper bound in very few number of iterations and therefore the algorithm converges very quickly. An example of evolution the objective function in Model I can be found in appendixD.2. A possible drawback come from stagnation in local maxima because it sometimes happen that algorithm finds some suboptimal solution and then it is not able to move to wherever else. This drawback can be solved by repetition of the algorithm or increasing a number of mutations. Mostly the found suboptimal solution is not very far from the optimal one and therefore it can be considered as a solution with certain accuracy.

5 Conclusion

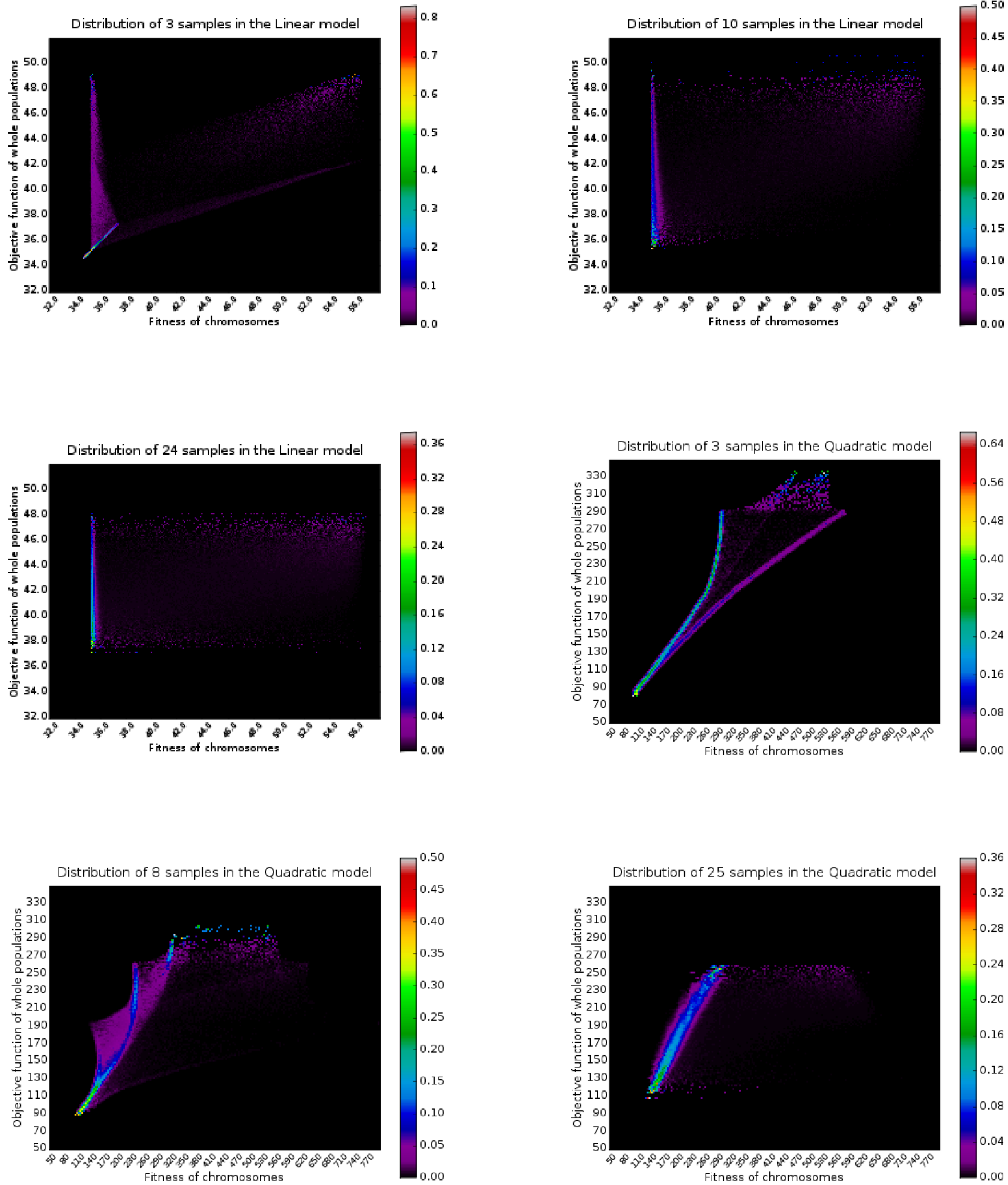
The goal of this diploma thesis was to introduce problems of stochastic programming and to show how to solve them by modern heuristic methods. As a particular problem has been chosen a problem of determining boundaries for an objective function in mathematical models influenced by random variables. We have developed a special computer program based on genetic algorithm [1] and it has been tested and adjusted for particular linear and quadratic mathematical models. At first, each model has been investigated by classical Monte Carlo method [3] and determined theoretical boundaries for each of them. The main idea of an applying genetic algorithms is to consider a population of chromosomes as a representation for probability distribution occurring in the given models and for obtaining an extreme values of their objective functions we need to find the distribution which cause it. The algorithm has been divided into two separate parts. The tested models contain from 2 to 4 randomnesses defined as independent as well dependent random variables on convex and non-convex sets.

A dealing with a problem of determining the minimal objective function have led us to introduce a special mutation operator, called *Touch*, which is designed for little modifications in genetic information contrary to usual mutation which can change chromosomes very dramatically. The results obtained from this implementation are very promising because we are able to find the minimum in just a few iterations. A determining the maximum value involved a developing of a new kind of operator for integration offsprings into a population which removes chromosomes with average fitness and therefore the population keeps big diversity. This operator has been called *Average Out* and together with aforementioned operator Touch gives very good results. The probability distributions with the maximal objective function are often concentrated in a few discrete points and our algorithm has shown it is able to find them and stay very close to that solution up to end of the running.

So the developed implementation of the algorithm works successfully for all of the tested models and converges very quickly even for small populations. It is very important to have an algorithm which does not need big populations because in the opposite case the evaluating can take a lot of computing time and therefore it can become useless. The main conclusion of the thesis is that we have found out the algorithm based on idea of approximating the probability distribution by a population of chromosomes can give very good results and it can save a significant amount of the expensive computing resources. Moreover the boundaries determined by classical Monte Carlo method give us just a certain confidence interval whereas the boundaries obtained from our algorithm can find almost the real extreme values and therefore it can describe the model in much more accuracy way.

A Distributions of chromosome fitnesses

Distributions of particular chromosome fitnesses in popultions. The precise description can be found in section 4.3.



B Data for evaluating populations

B.1 Model I

These tables contains data from the test in section 4.4 for Model I. Each row symbolize a population with given number n of chromosomes and columns symbolize number of new chromosomes added to the population. Data in the "ord" column represent ratio between all chromosomes and those whose order was changed just by one place or stayed unchanged. The "fit" column gives ratio for chromosomes whose fitness variation is in interval $(-0.25, 0.25)$.

	1		2		3		4		5		6	
n	ord	fit	ord	fit	ord	fit	ord	fit	ord	fit	ord	fit
2	0.98	0.71										
3	0.99	0.90	0.98	0.84								
4	0.99	0.96	0.98	0.93	0.98	0.92						
5	1.00	0.98	0.99	0.95	0.98	0.96	0.97	0.94				
6	0.99	0.97	0.99	0.97	0.99	0.97	0.98	0.96	0.98	0.96		
7	0.99	0.98	0.98	0.96	0.98	0.96	0.98	0.96	0.98	0.95	0.97	0.95
8	0.99	0.98	0.98	0.98	0.98	0.97	0.98	0.97	0.98	0.97	0.97	0.96
9	1.00	0.99	0.99	0.98	0.99	0.97	0.98	0.97	0.97	0.97	0.97	0.97
10	1.00	0.99	0.99	0.98	0.99	0.98	0.98	0.98	0.98	0.97	0.98	0.98
11	1.00	1.00	0.99	0.99	0.99	0.99	0.98	0.99	0.98	0.98	0.98	0.98
12	1.00	0.99	0.99	0.99	0.99	0.99	0.98	0.99	0.98	0.99	0.98	0.98
13	0.99	1.00	0.98	0.99	0.98	0.99	0.98	0.98	0.98	0.99	0.98	0.98
14	0.99	0.99	0.99	0.99	0.98	0.99	0.98	0.99	0.97	0.98	0.98	0.99
15	0.99	1.00	0.99	0.99	0.99	0.99	0.98	0.98	0.98	0.99	0.97	0.98
16	0.99	1.00	0.99	0.99	0.98	0.99	0.98	0.99	0.98	0.99	0.97	0.99
17	1.00	1.00	0.99	1.00	0.99	0.99	0.99	0.99	0.98	0.99	0.98	0.99
18	1.00	1.00	0.99	1.00	0.99	0.99	0.99	0.99	0.98	1.00	0.98	0.99
19	1.00	1.00	0.99	0.99	0.99	0.99	0.99	0.99	0.98	0.99	0.98	1.00
20	1.00	1.00	0.99	1.00	0.99	0.99	0.99	1.00	0.98	0.99	0.98	0.99

	7		8		9		10		11		12	
n	ord	fit	ord	fit	ord	fit	ord	fit	ord	fit	ord	fit
8	0.96	0.97										
9	0.98	0.98	0.97	0.97								
10	0.98	0.98	0.98	0.98	0.98	0.97						
11	0.98	0.98	0.98	0.98	0.97	0.98	0.98	0.98				
12	0.98	0.99	0.98	0.98	0.98	0.98	0.98	0.98	0.97	0.98		
13	0.98	0.99	0.98	0.99	0.97	0.98	0.97	0.98	0.97	0.98	0.96	0.98
14	0.97	0.98	0.98	0.99	0.97	0.99	0.96	0.98	0.96	0.98	0.96	0.98
15	0.98	0.98	0.97	0.98	0.97	0.98	0.97	0.98	0.97	0.98	0.96	0.98
16	0.98	0.99	0.97	0.99	0.97	0.98	0.97	0.99	0.96	0.98	0.97	0.98
17	0.98	0.99	0.98	0.99	0.98	0.99	0.97	0.99	0.97	0.98	0.97	0.99
18	0.98	0.99	0.98	0.99	0.98	0.99	0.98	0.99	0.97	0.99	0.97	0.99
19	0.98	0.99	0.98	0.99	0.98	0.99	0.98	0.99	0.97	0.99	0.97	0.99
20	0.98	0.99	0.98	0.99	0.98	0.99	0.97	0.99	0.97	0.99	0.97	0.99

	13		14		15		16		17		18		19	
n	ord	fit	ord	fit	ord	fit	ord	fit	ord	fit	ord	fit	ord	fit
14	0.96	0.98												
15	0.96	0.98	0.96	0.98										
16	0.96	0.98	0.96	0.98	0.96	0.98								
17	0.96	0.99	0.97	0.99	0.96	0.99	0.96	0.98						
18	0.97	0.99	0.97	0.99	0.97	0.99	0.97	0.99	0.96	0.99				
19	0.97	0.99	0.97	0.99	0.97	0.99	0.97	0.99	0.96	0.99	0.96	0.99		
20	0.97	0.99	0.97	0.99	0.96	0.99	0.96	0.99	0.96	0.98	0.96	0.99	0.96	0.99

B.2 Model IV

In the tables for Model IV the "ord" column gives ratio for variation in interval $(-1,1)$ and the "fit" column is ratio with respect to chromosomes with fitness variation in interval $(-25,25)$

	1		2		3		4		5		6	
n	ord	fit	ord	fit	ord	fit	ord	fit	ord	fit	ord	fit
2	0.99	0.64										
3	0.99	0.62	0.96	0.50								
4	0.98	0.68	0.95	0.56	0.95	0.58						
5	0.98	0.76	0.97	0.63	0.95	0.59	0.93	0.63				
6	0.99	0.86	0.96	0.72	0.95	0.61	0.95	0.64	0.93	0.58		
7	0.98	0.89	0.97	0.74	0.95	0.70	0.95	0.67	0.93	0.66	0.92	0.62
8	0.98	0.92	0.96	0.79	0.95	0.75	0.95	0.70	0.93	0.67	0.93	0.66
9	0.98	0.90	0.96	0.84	0.96	0.75	0.94	0.73	0.94	0.70	0.94	0.71
10	0.98	0.92	0.96	0.85	0.95	0.81	0.95	0.75	0.95	0.73	0.94	0.74
11	0.98	0.94	0.97	0.89	0.96	0.84	0.95	0.77	0.95	0.79	0.93	0.75
12	0.98	0.95	0.96	0.92	0.96	0.84	0.95	0.79	0.94	0.79	0.94	0.77
13	0.98	0.97	0.97	0.94	0.95	0.83	0.96	0.82	0.95	0.83	0.93	0.79
14	0.98	0.97	0.97	0.94	0.96	0.91	0.95	0.85	0.94	0.83	0.93	0.79
15	0.98	0.97	0.97	0.95	0.96	0.91	0.96	0.90	0.95	0.87	0.94	0.83
16	0.98	0.99	0.97	0.97	0.96	0.93	0.95	0.91	0.94	0.86	0.94	0.84
17	0.98	0.98	0.97	0.96	0.96	0.94	0.96	0.91	0.95	0.88	0.94	0.86
18	0.98	0.98	0.97	0.98	0.96	0.95	0.96	0.91	0.95	0.91	0.94	0.89
19	0.98	0.99	0.97	0.98	0.96	0.95	0.95	0.95	0.95	0.91	0.95	0.89
20	0.98	0.99	0.97	0.98	0.96	0.96	0.96	0.93	0.95	0.92	0.94	0.90

	7		8		9		10		11		12	
n	ord	fit	ord	fit	ord	fit	ord	fit	ord	fit	ord	fit
8	0.92	0.63										
9	0.93	0.67	0.92	0.63								
10	0.92	0.70	0.92	0.70	0.90	0.65						
11	0.93	0.70	0.92	0.69	0.91	0.68	0.90	0.67				
12	0.93	0.75	0.92	0.74	0.92	0.73	0.90	0.70	0.90	0.68		
13	0.92	0.76	0.92	0.77	0.92	0.73	0.92	0.72	0.90	0.72	0.90	0.71
14	0.92	0.79	0.93	0.78	0.92	0.77	0.90	0.74	0.89	0.73	0.90	0.73
15	0.93	0.79	0.93	0.83	0.92	0.79	0.91	0.79	0.89	0.76	0.90	0.78
16	0.93	0.84	0.93	0.83	0.92	0.79	0.91	0.77	0.91	0.80	0.91	0.81
17	0.94	0.83	0.93	0.83	0.92	0.83	0.91	0.82	0.91	0.80	0.90	0.79
18	0.93	0.86	0.92	0.84	0.92	0.86	0.91	0.84	0.92	0.82	0.91	0.81
19	0.93	0.89	0.93	0.85	0.93	0.86	0.92	0.85	0.91	0.83	0.91	0.85
20	0.94	0.90	0.93	0.86	0.92	0.87	0.92	0.86	0.91	0.83	0.90	0.82

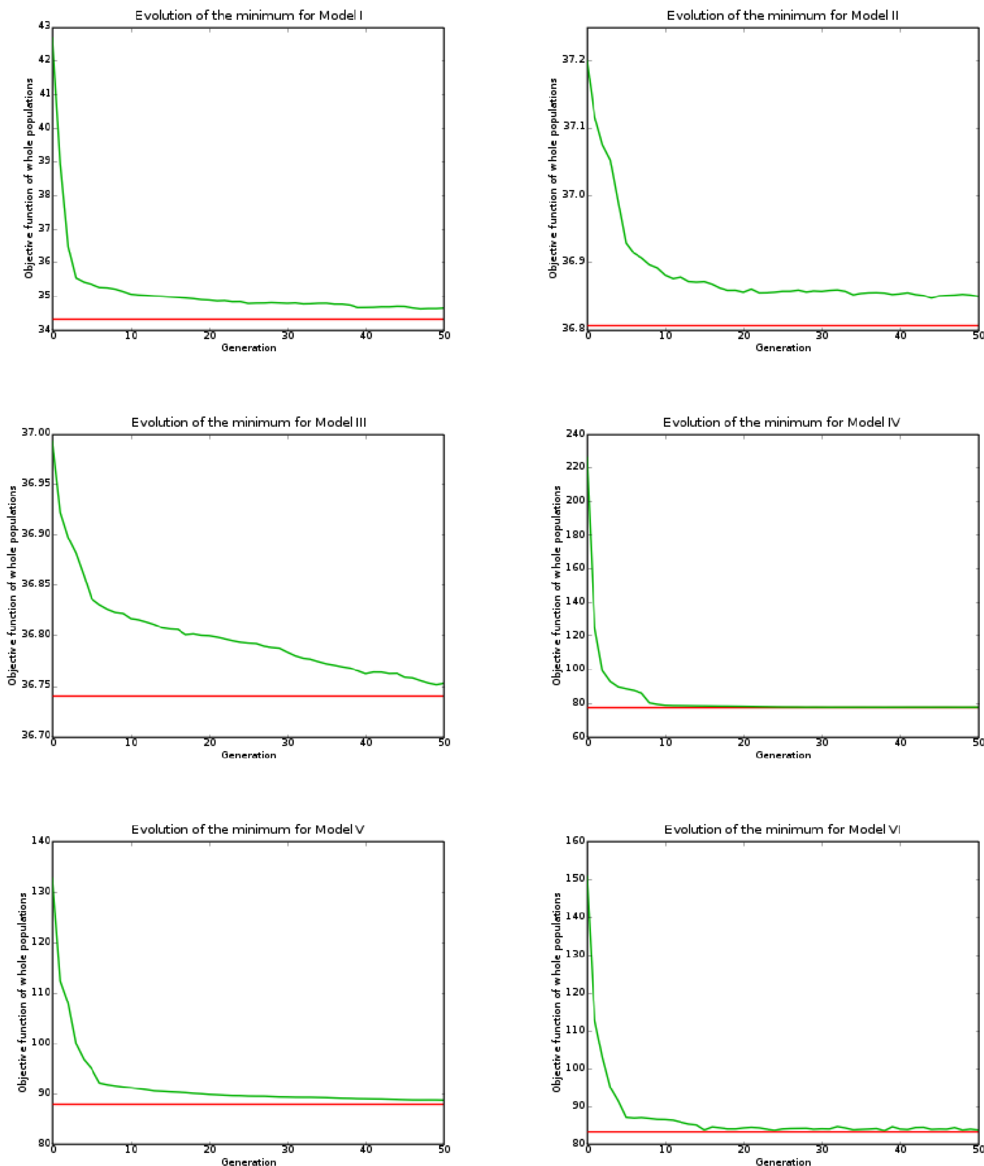
	13		14		15		16		17		18		19	
n	ord	fit	ord	fit	ord	fit	ord	fit	ord	fit	ord	fit	ord	fit
14	0.89	0.70												
15	0.90	0.74	0.89	0.73										
16	0.90	0.77	0.88	0.74	0.89	0.75								
17	0.90	0.80	0.91	0.76	0.90	0.78	0.87	0.75						
18	0.90	0.79	0.89	0.78	0.89	0.80	0.86	0.74	0.88	0.77				
19	0.89	0.83	0.89	0.80	0.89	0.82	0.89	0.82	0.88	0.79	0.87	0.77		
20	0.89	0.82	0.89	0.81	0.90	0.81	0.89	0.80	0.88	0.82	0.87	0.80	0.87	0.78

C Searching for a minimum

There are examples of the algorithm behaviour during searching for the minimal objective function value. All the data in this chapter are related to the section 4.6.

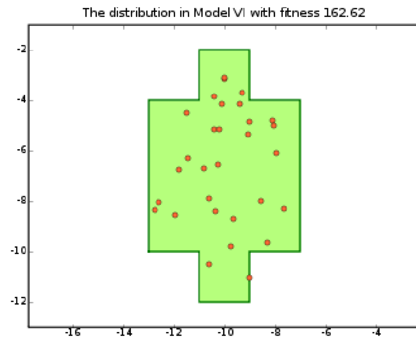
C.1 Evolution of the objective function

These examples decribe an evolution of the objective function for all the tested models.

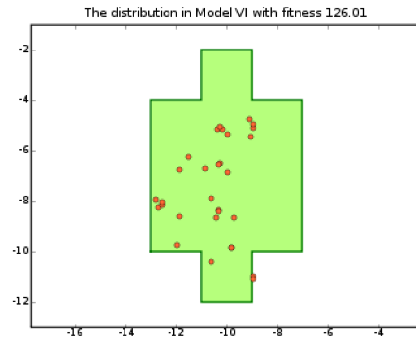


C.2 Evolution of distributions

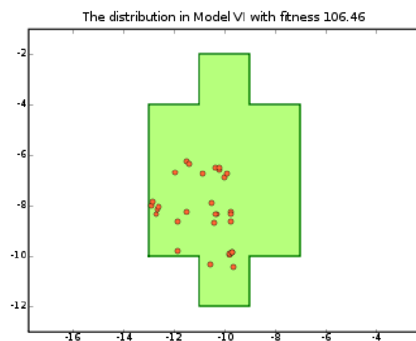
The examples in this section show distributions of chromosomes during a searching for the minimal objective function value in Model VI.



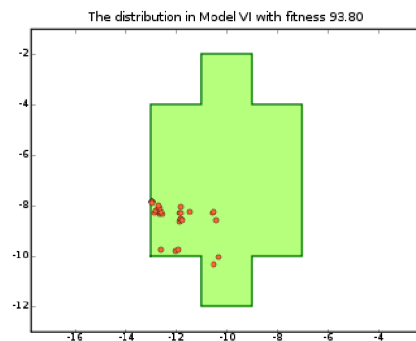
1st generation



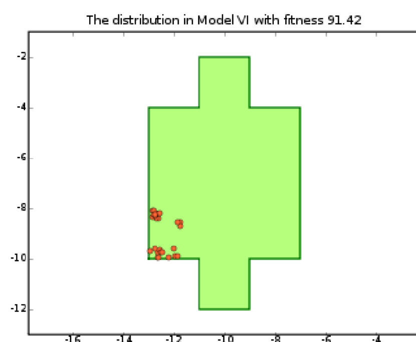
2nd generation



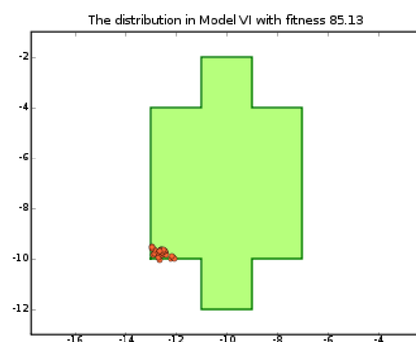
3rd generation



5th generation



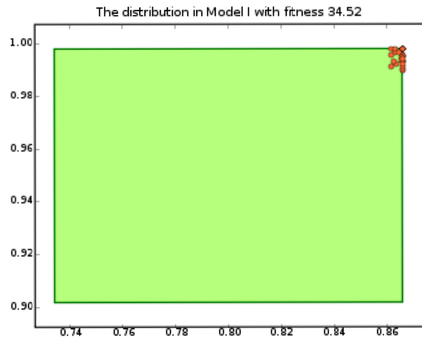
7th generation

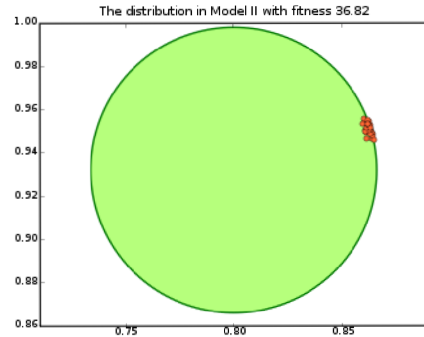


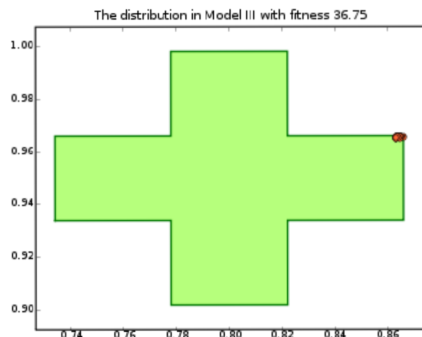
10th generation

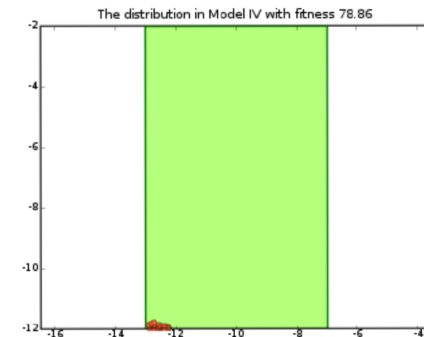
C.3 Final distributions with minimal value

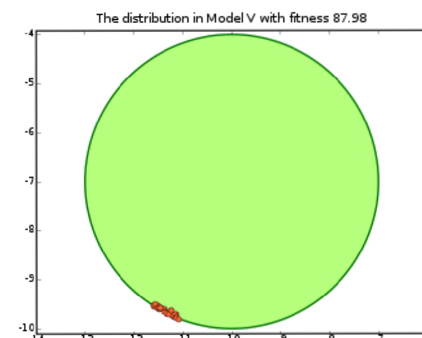
There are examples of distributions with lower objective function found by our implementation of a genetic algorithm.

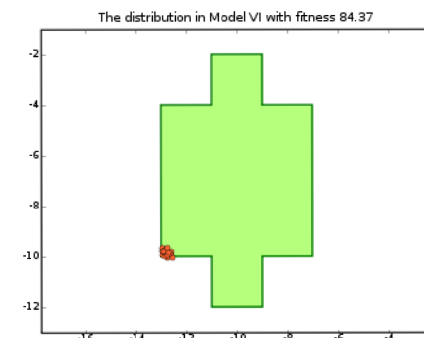








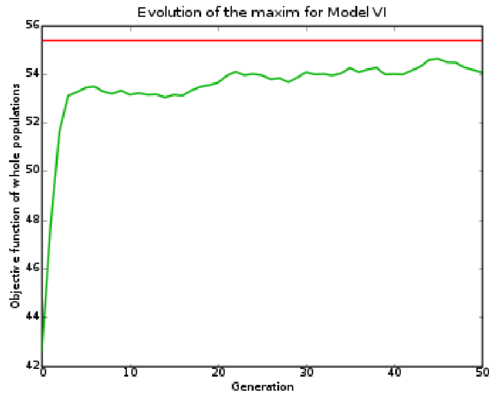


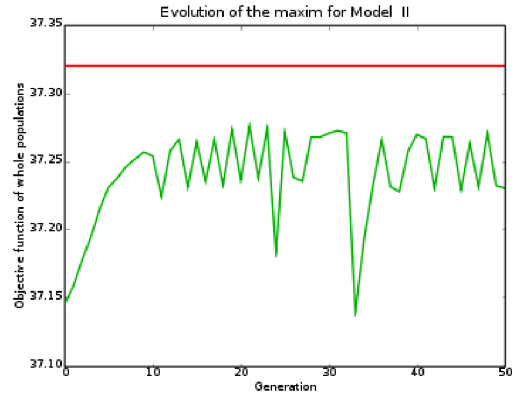


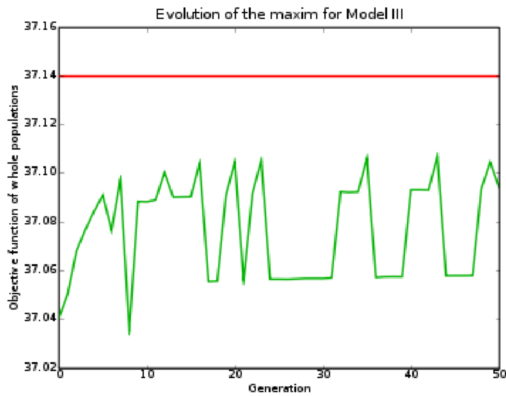
D Searching for the maximum

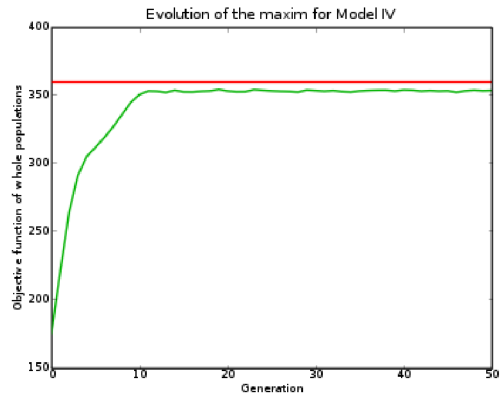
D.1 Evolution of the searching for maximal value

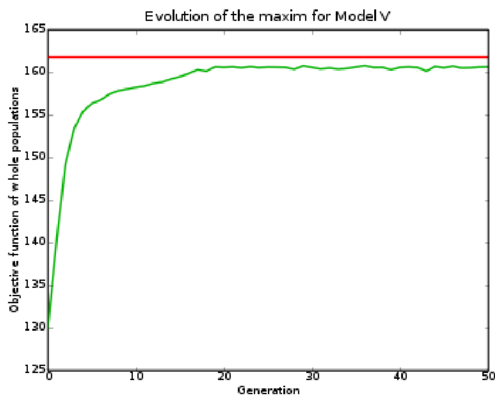
These examples decribe evolution of the objective function during searching for maximal values. (All data in this appendix are related to the section 4.7)

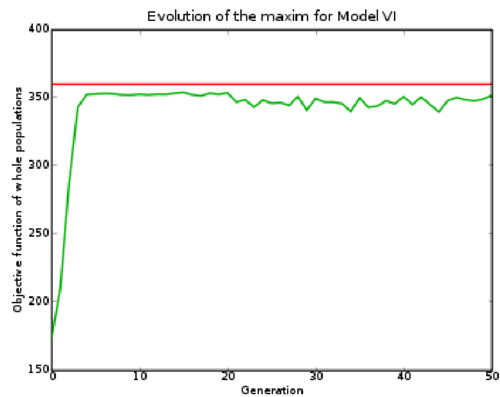






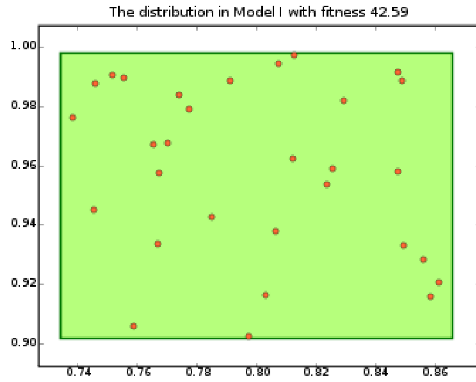




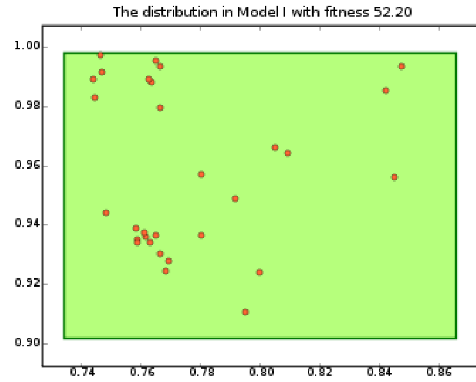


D.2 Evolution of maximal distributions

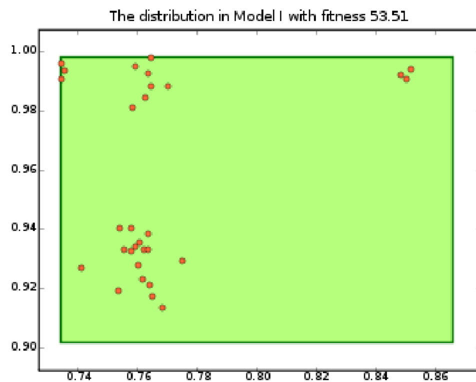
Examples of evolution of chromosomes during a searching for the maximum in Model I.



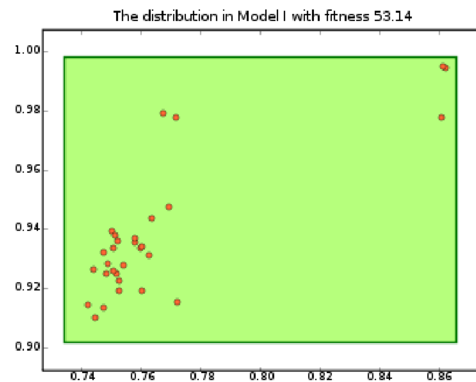
1st generation



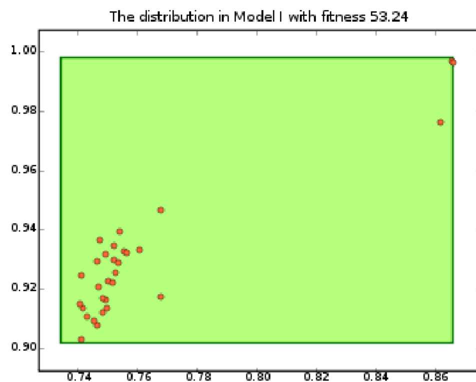
6th generation



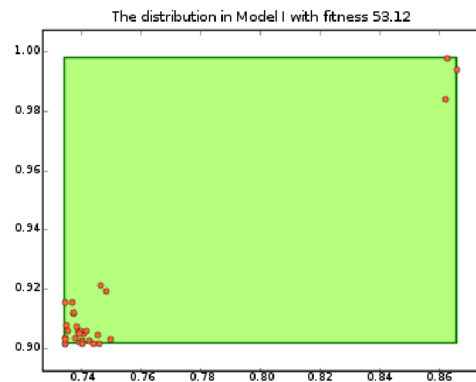
11th generation



19th generation



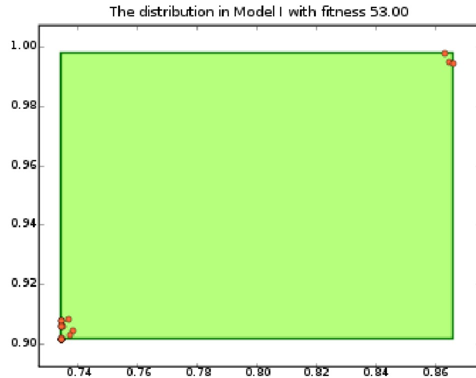
25th generation

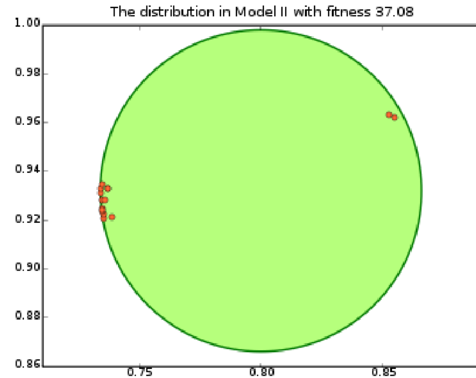


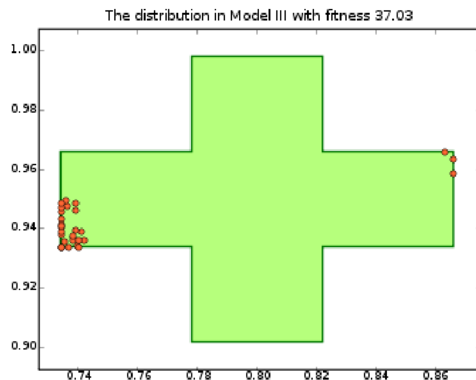
30th generation

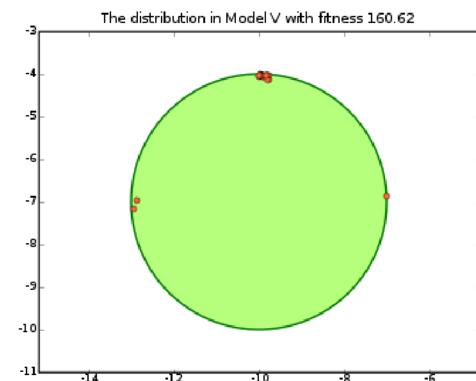
D.3 Final distributions with maximal value

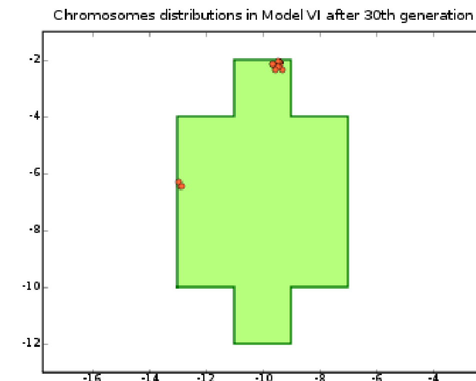
Examples of distributions with high objective function found by genetic algorithm.











Index

a
algorithm
  deterministic *15*
  simplex *2*
  stochastic *16*
algorithms
  evolutionary *17*
approximating problem *8, 9*

b
brute force *14*

c
candidate solution *1, 10*
chromosome *20*
complete fixed recourse *6*
confidence coefficient *9*
confidence interval *9*
confidence level *9*
constraint *1*
convexity
  see convex polyhedron
convex polyhedron *2, 3*

d
deterministic equivalent *8*

f
feasible set *1*
fitness *21*
fitness function *20*
fitness of a population *29*
fixed recourse matrix *6*

g
genes *20, 23*
genetic algorithm *19*
gradient ascent *15*

h
Hamming barrier *23*
here-and-now *5*
heuristic *14*
hill climbing *15*

i
individual *20*
integration
  Average Out *47*

l
local search *15*

m
method
  Monte Carlo *8*
minimax approach *12*
model
  deterministic *4*
  linear
    see linear programming
  multi-stage *5*
  stochastic
    see stochastic programming
  two-stage *5*

n
normalized contribution *21*

o
objective function *1*
offspring *20*
operator
  crossover *20*
  deletion *20*
  mutation *20*
  parent selection *20*
optimality gap *10*

optimization *1*
  ant colony *16*
  bees *17*

p
population *19*
programming
  binary *2*
  integer *2*
  linear *2*
  linear stochastic *6*
  mathematical
    see optimization
  quadratic *3*
  stochastic *4*

r
recourse *5*

s
scenario *8*
search space *1*
shadow zone *23*
simulated annealing *16*
stage
  first *5*
  second *5*

t
tabu search *15*
terminate criterion *20*
touch
  mutation *44*
travelling salesman problem *14*

w
wait-and-see *5*

References

[1] J. Roupec and P. Popela, *Scenario generation and analysis by heuristic algorithms* (2007)
[2] J. Roupec, PhD Dissertation, VUT Brno, Fakulta strojního inženýrství, (2001).
[3] W. K. Mak, D. P. Morton, and R. K. Wood, *Monte carlo bounding techniques for determining solution quality in stochastic programs*, Operations Research Letters
[4] J. Dupačová, *Stochastické programování*. (Československá redakce VN MON, 1986).
[5] S. W. Wallace and P. Kall, *Stochastic Programming*. (John Wiley & Sons, Inc., 1994).
[6] O. online documentation http://scipy.org/scipy/scikits/wiki/OpenOpt.
[7] C. R. Reeves, editor, *Modern heuristic techniques for combinatorial problems* (John Wiley & Sons, Inc., New York, NY, USA, 1993).
[8] L. D. Chambers, *Practical Handbook of Genetic Algorithms*. (CRC Press, Inc., Boca Raton, FL, USA, 1995).
[9] L. D. Chambers, *The Practical Handbook of Genetic Algorithms: Applications, Second Edition*. (CRC Press, Inc., Boca Raton, FL, USA, 2000).
[10] M. Dorigo and C. Blum, *Ant colony optimization theory: a survey*, Theor. Comput. Sci. **344** (2005), no. 2-3, 243–278
[11] M. Dorigo, G. D. Caro, and L. M. Gambardella, *Ant algorithms for discrete optimization*, Artif. Life **5** (1999), no. 2, 137–172
[12] F. Glover and F. Laguna, *Tabu Search*. (Kluwer Academic Publishers, Norwell, MA, USA, 1997).
[13] F. Glover, E. Taillard, and D. de Werra, *A user's guide to tabu search*, Ann. Oper. Res. **41** (1993), no. 1-4, 3–28
[14] Z. Michalewicz and D. B. Fogel, *How to solve it: modern heuristics*. (Springer-Verlag New York, Inc., New York, NY, USA, 2000).
[15] P. J. M. Laarhoven and E. H. L. Aarts, editors, *Simulated annealing: theory and applications* (Kluwer Academic Publishers, Norwell, MA, USA, 1987).
[16] P. Tarasewich and P. R. McMullen, *Swarm intelligence: power in numbers*, Commun. ACM **45** (2002), no. 8, 62–67
[17] S. A. Tarim, S. Manandhar, and T. Walsh, *Stochastic constraint programming: A scenario-based approach*, Constraints **11** (2006), no. 1, 53–80
[18] A. Eichhorn and W. Römisch, *Stochastic integer programming: Limit theorems and confidence intervals*, Math. Oper. Res. **32** (2007), no. 1, 118–135
[19] E. Polak and J. O. Royset, *Efficient sample sizes in stochastic nonlinear programming*, J. Comput. Appl. Math. **217** (2008), no. 2, 301–310
[20] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. (Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989).
[21] A. Schrijver, *Theory of linear and integer programming*. (John Wiley & Sons, Inc., New York, NY, USA, 1986).
[22] J. Dupačová, J. Hurt, and J. Štěpán, *Stochastic Modeling in Ecomocics and Finance*. (Kluwer Academic Publisher, 2003).